

ĐẠI HỌC BÁCH KHOA HÀ NỘI

ĐỒ ÁN TỐT NGHIỆP

**Xây dựng hệ thống thu thập, xử lý, lưu trữ
và trực quan hóa dữ liệu lớn - Thử nghiệm với dữ liệu
các chuyến taxi ở thành phố New York**

ĐÀO ANH QUÂN

quan.da215631@sis.hust.edu.vn

Chương trình đào tạo: Kỹ thuật Máy tính

Giảng viên hướng dẫn: TS. Đinh Thị Hà Ly

Chữ kí GVHD

Khoa: Kỹ thuật máy tính

Trường: Công nghệ Thông tin và Truyền thông

HÀ NỘI, 06/2025

LỜI CAM KẾT

Họ và tên sinh viên: Đào Anh Quân

Điện thoại liên lạc: 0972833225

Email: quan.da215631@sis.hust.edu.vn

Lớp: Kỹ thuật máy tính 02 - K66

Hệ đào tạo: Cử nhân

Tôi – *Đào Anh Quân* – cam kết Đồ án Tốt nghiệp (ĐATN) là công trình nghiên cứu của bản thân tôi dưới sự hướng dẫn của TS. *Đinh Thị Hà Ly*. Các kết quả nêu trong ĐATN là trung thực, là thành quả của riêng tôi, không sao chép theo bất kỳ công trình nào khác. Tất cả những tham khảo trong ĐATN – bao gồm hình ảnh, bảng biểu, số liệu, và các câu từ trích dẫn – đều được ghi rõ ràng và đầy đủ nguồn gốc trong danh mục tài liệu tham khảo. Tôi xin hoàn toàn chịu trách nhiệm với dù chỉ một sao chép vi phạm quy chế của nhà trường.

Hà Nội, ngày tháng năm

Tác giả ĐATN

Đào Anh Quân

LỜI CẢM ƠN

Vậy là chặng đường 4 năm gắn bó với Bách Khoa đã kết thúc, để lại cho em biết bao bài học, kỷ niệm đáng nhớ. Để có thể đi hết chặng đường này, em đã được sự giúp đỡ không nhỏ của thầy cô, bạn bè và gia đình.

Trước tiên em muốn gửi lời cảm ơn sâu sắc nhất đến TS. Đinh Thị Hà Ly - giảng viên hướng dẫn của em trong quá trình thực hiện đề án tốt nghiệp. Mặc dù cô không phải là giáo viên em đăng ký lúc đầu nhưng nhờ sự hướng dẫn vô cùng tận tình, tâm huyết của cô mà em đã có thể hoàn thành được đề án tốt nghiệp của mình.

Con cảm ơn bố mẹ vì đã luôn đồng hành với con, luôn ở bên con lúc khó khăn nhất, là chỗ dựa tinh thần, động lực cố gắng của con. Cảm ơn những người bạn đã luôn giúp đỡ, đồng hành với tôi trong suốt thời gian học đại học.

Và cuối cùng, cảm ơn bản thân vì đã luôn kiên trì, nỗ lực, không bỏ cuộc trước mọi khó khăn. Dù kết quả có ra sao thì hãy luôn tự hào vì chính bản thân mình, tự hào vì những gì mình đã làm được. Và hãy nhớ rằng: Mọi chuyện rồi cũng sẽ ổn thôi.

Xin chân thành cảm ơn!

TÓM TẮT NỘI DUNG ĐỒ ÁN

Trong thời đại bùng nổ của khoa học và công nghệ, dữ liệu trở thành một loại tài nguyên vô cùng quan trọng, có tiềm năng lớn trong việc phân tích và ra quyết định. Sự phát triển mạnh mẽ của Internet, thiết bị di động và các hệ thống cảm biến đã góp phần tạo ra lượng dữ liệu khổng lồ mỗi ngày, bao phủ hầu hết mọi lĩnh vực từ kinh tế, y tế, giáo dục cho đến giao thông và thương mại điện tử. Để đối mặt với thực trạng này, nhiều công nghệ, kỹ thuật và nền tảng hiện đại đã được phát triển nhằm hỗ trợ việc thu thập, xử lý và lưu trữ dữ liệu lớn, như các cơ sở dữ liệu NoSQL, các công nghệ hỗ trợ tính toán song song, lưu trữ phân tán... Những giải pháp này đã và đang được ứng dụng rộng rãi, góp phần giải quyết phần nào các bài toán liên quan đến dữ liệu lớn. Tuy nhiên, đây vẫn là một vấn đề đầy thách thức, đòi hỏi những công nghệ và hướng tiếp cận phù hợp để đáp ứng hiệu quả yêu cầu về quy mô và tốc độ xử lý.

Đồ án này hướng tới việc xây dựng một hệ thống đầy đủ các quá trình từ thu thập, xử lý, lưu trữ đến trực quan hóa dữ liệu lớn, và được thử nghiệm trên dữ liệu các chuyến taxi của thành phố New York, Mỹ. Cụ thể, hệ thống xây dựng theo kiến trúc Lambda, áp dụng những công nghệ hiện đại như Spark, Kafka, Hadoop... nhằm xử lý đồng thời cả dữ liệu theo luồng và theo lô. Toàn bộ hệ thống được triển khai trên một cụm Kubernetes thực tế, được xây dựng thủ công từ các máy chủ Ubuntu, thay vì sử dụng các giải pháp mô phỏng như Minikube. Cách triển khai này đảm bảo hệ thống vận hành trong môi trường sát thực tế, có khả năng mở rộng và kiểm soát linh hoạt các thành phần hạ tầng.

Kết quả, đồ án đã xây dựng thành công một pipeline xử lý dữ liệu lớn hoàn chỉnh với các mô đun riêng biệt có khả năng mở rộng, chịu lỗi cao, tự động hóa các quy trình, cho phép phân tích dữ liệu một cách linh hoạt. Hệ thống có tiềm năng ứng dụng trong các môi trường sản xuất thực tế, làm nền tảng cho các bài toán phân tích dữ liệu lớn trong tương lai.

Sinh viên thực hiện

(Ký và ghi rõ họ tên)

ABSTRACT

In the era of booming science and technology, data has become an extremely important resource with great potential in analysis and decision-making. The rapid development of the Internet, mobile devices, and sensor systems has contributed to creating enormous amounts of data every day, covering almost all fields from economics, healthcare, and education to transportation and e-commerce. To address this situation, many modern technologies, techniques, and platforms have been developed to support the collection, processing, and storage of big data, such as NoSQL databases, technologies supporting parallel computing, and distributed storage, among others. These solutions have been widely applied, helping to resolve some of the challenges related to big data. However, this remains a challenging issue that requires appropriate technologies and approaches to effectively meet the demands of scale and processing speed.

This thesis aims to build a comprehensive system for the entire process of collecting, processing, storing, and visualizing big data, tested on taxi trip data from New York City, USA. Specifically, the system is designed based on the Lambda architecture, utilizing modern technologies such as Spark, Kafka, Hadoop, etc., to handle both streaming and batch data simultaneously. The entire system is deployed on a real Kubernetes cluster, manually built from Ubuntu servers, rather than using simulation solutions like Minikube. This deployment approach ensures that the system operates in an environment close to real-world conditions, with scalable and flexible infrastructure management.

As a result, the project successfully developed a complete big data processing pipeline with modular components that are scalable, highly fault-tolerant, and automated, enabling flexible data analysis. The system has potential applications in real-world production environments and can serve as a foundation for future big data analytics tasks.

MỤC LỤC

CHƯƠNG 1. GIỚI THIỆU ĐỀ TÀI.....	1
1.1 Đặt vấn đề.....	1
1.2 Mục tiêu và phạm vi đề tài.....	2
1.3 Định hướng giải pháp.....	3
1.4 Bố cục đồ án	4
CHƯƠNG 2. KHẢO SÁT VÀ PHÂN TÍCH YÊU CẦU.....	5
2.1 Khảo sát hiện trạng	5
2.1.1 Khảo sát về kiến trúc hệ thống.....	5
2.1.2 Khảo sát về môi trường triển khai	6
2.2 Tổng quan chức năng	8
2.2.1 Yêu cầu chức năng	8
2.2.2 Yêu cầu phi chức năng.....	9
CHƯƠNG 3. CÔNG NGHỆ SỬ DỤNG.....	11
3.1 Công nghệ lập lịch, điều phối - Apache Airflow	11
3.2 Công nghệ thu thập dữ liệu - Apache Kafka	11
3.3 Công nghệ xử lý dữ liệu - Apache Spark	12
3.3.1 Spark SQL	12
3.3.2 Spark MLlib	12
3.3.3 Spark Structured Streaming.....	12
3.4 Công nghệ lưu trữ dữ liệu	13
3.4.1 Hadoop HDFS	13
3.4.2 Redis	13
3.4.3 PostgreSQL.....	14

3.5 Công nghệ trực quan hóa dữ liệu	14
3.5.1 Superset	14
3.5.2 Spring Boot và ReactJS	15
3.6 Công nghệ giám sát hệ thống	15
3.6.1 Prometheus.....	15
3.6.2 Grafana	16
3.7 Công nghệ triển khai, điều phối - Kubernetes	16
CHƯƠNG 4. THIẾT KẾ, TRIỂN KHAI VÀ KẾT QUẢ THỰC NGHIỆM	
18	
4.1 Thiết kế kiến trúc.....	18
4.1.1 Kiến trúc tổng quan hệ thống	18
4.1.2 Mô-đun thu thập dữ liệu	19
4.1.3 Mô-đun xử lý dữ liệu.....	21
4.1.4 Mô-đun lập lịch, tự động.....	24
4.1.5 Mô-đun lưu trữ dữ liệu	27
4.1.6 Mô-đun trực quan hóa dữ liệu	30
4.2 Triển khai hệ thống	32
4.2.1 Quy trình và cách thức triển khai ứng dụng trên Kubernetes.....	32
4.2.2 Kết quả triển khai.....	33
4.3 Kết quả thực nghiệm	36
4.3.1 Kết quả thu thập dữ liệu.....	36
4.3.2 Kết quả xử lý dữ liệu	38
4.3.3 Kết quả lưu trữ dữ liệu.....	39
4.3.4 Kết quả trực quan hóa dữ liệu.....	40
4.3.5 Khả năng chịu lỗi.....	42
4.3.6 Khả năng mở rộng.....	43

CHƯƠNG 5. CÁC GIẢI PHÁP VÀ ĐÓNG GÓP NỔI BẬT	45
5.1 Giới thiệu vấn đề	45
5.2 Thiết kế kiến trúc môi trường triển khai Kubernetes.....	45
5.2.1 Cụm máy chủ Kubernetes	45
5.2.2 Giao diện quản lý - Rancher	46
5.2.3 Thành phần cân bằng tải - Nginx	48
5.2.4 NFS Server - Lưu trữ dữ liệu	49
5.3 Kết quả đạt được.....	50
5.3.1 Triển khai cụm máy chủ.....	50
5.3.2 Lưu trữ dữ liệu các thành phần	51
5.3.3 Công cụ giám sát & Sao lưu hệ thống	51
CHƯƠNG 6. KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN	54
6.1 Kết luận	54
6.2 Hướng phát triển.....	55
TÀI LIỆU THAM KHẢO.....	57

DANH MỤC HÌNH VẼ

Hình 2.1	Yêu cầu chức năng của hệ thống	8
Hình 4.1	Kiến trúc tổng quan của hệ thống	18
Hình 4.2	Một bản ghi được gửi lên Kafka	21
Hình 4.3	Quá trình xử lý dữ liệu theo luồng	22
Hình 4.4	Quá trình xử lý dữ liệu theo lô	23
Hình 4.5	Đồ thị DAG cho quá trình xử lý streaming	24
Hình 4.6	Đồ thị DAG cho quá trình xử lý theo lô	25
Hình 4.7	Kết nối Airflow với Spark để sử dụng SparkSubmitOperator .	27
Hình 4.8	Dữ liệu trên kênh realtime-trip-channel	28
Hình 4.9	Tạo biểu đồ trong Superset	31
Hình 4.10	Các Pod của hệ thống	34
Hình 4.11	Các Deployment của hệ thống	34
Hình 4.12	Các StatefulSet của hệ thống	35
Hình 4.13	Các Service của hệ thống	36
Hình 4.14	Dữ liệu giả lập streaming qua API	37
Hình 4.15	Gửi dữ liệu qua Kafka	37
Hình 4.16	Xử lý dữ liệu theo lô bằng Spark SQL	38
Hình 4.17	Lập lịch tác vụ trên Airflow	38
Hình 4.18	Cấu trúc thư mục trên HDFS	39
Hình 4.19	Cấu trúc bảng trong PostgreSQL	40
Hình 4.20	Tạo báo cáo bằng Superset	41
Hình 4.21	Biểu đồ số chuyển đi theo thời gian thực	41
Hình 4.22	Thống kê tuyến đường phổ biến theo địa điểm	42
Hình 4.23	Thống kê dữ liệu theo tháng	42
Hình 4.24	Khả năng chịu lỗi của cụm Kafka	43
Hình 5.1	Kiến trúc của một cụm Kubernetes	46
Hình 5.2	Giao diện quản lý của Rancher	47
Hình 5.3	Các dịch vụ được truy cập qua Nginx-Ingress	48
Hình 5.4	Các Persistent Volume Claim của cụm Kafka	49
Hình 5.5	Cấu trúc thư mục trên NFS Server	51
Hình 5.6	Giao diện Grafana giám sát tài nguyên của các Pod	52
Hình 5.7	Dữ liệu sao lưu cụm K8s lưu trữ trên MinIO	53

DANH MỤC BẢNG BIỂU

Bảng 2.1	Các mô hình triển khai hệ thống dữ liệu lớn [4]	7
Bảng 4.1	Thông tin của các trường dữ liệu thu thập	20
Bảng 4.2	Các trường dữ liệu được thêm trong bảng <code>fact_trips</code> . . .	29
Bảng 4.3	Các trường dữ liệu của bảng <code>analyze_by_time</code>	30
Bảng 4.4	Các trường dữ liệu của bảng <code>analyze_by_routes</code>	30
Bảng 5.1	Thông số các máy ảo trong cụm	50

CHƯƠNG 1. GIỚI THIỆU ĐỀ TÀI

1.1 Đặt vấn đề

Trong kỷ nguyên số hiện nay, dữ liệu đang tăng trưởng với tốc độ chưa từng có, đặc biệt tại các đô thị lớn – nơi mọi hoạt động từ giao thông, thương mại đến hành chính đều được mã hóa thành dữ liệu. Ví dụ tại thành phố New York, mỗi ngày có hàng trăm nghìn chuyến taxi được ghi lại với đầy đủ thông tin về thời gian, địa điểm, quãng đường, chi phí và phương thức thanh toán. Tại Việt Nam, các hệ thống thu phí tự động trên cao tốc hay các ứng dụng giao đồ ăn như Grab, ShopeeFood cũng ghi nhận hàng triệu giao dịch mỗi ngày, bao gồm thông tin vị trí, thời gian giao hàng, món ăn, hình thức thanh toán và đánh giá của khách hàng. Trong lĩnh vực y tế, nhiều bệnh viện lớn đã bắt đầu số hóa hồ sơ khám chữa bệnh, kết quả xét nghiệm và hình ảnh chẩn đoán, tạo ra khối lượng dữ liệu khổng lồ mỗi ngày.

Tốc độ gia tăng và sự đa dạng của dữ liệu trong nhiều lĩnh vực đã vượt xa khả năng xử lý của các hệ thống truyền thống, dẫn đến sự hình thành và phát triển của khái niệm dữ liệu lớn (Big Data) – chỉ tập hợp dữ liệu có khối lượng lớn, tốc độ cao và đa dạng định dạng, đòi hỏi những công nghệ và kiến trúc chuyên biệt để thu thập, xử lý và khai thác hiệu quả. Những dữ liệu này, nếu được thu thập và xử lý một cách hiệu quả, có thể trở thành nền tảng quan trọng cho việc ra quyết định nhanh chóng, chính xác và khách quan. Thay vì dựa vào trực giác hay kinh nghiệm chủ quan, các tổ chức, doanh nghiệp và cơ quan quản lý có thể sử dụng dữ liệu để dự báo nhu cầu, tối ưu hóa nguồn lực, nâng cao trải nghiệm người dùng và phản ứng linh hoạt trước các tình huống phát sinh. Chẳng hạn, dữ liệu giao thông có thể giúp điều tiết đèn tín hiệu thông minh, giảm ùn tắc; dữ liệu mua sắm có thể hỗ trợ cá nhân hóa khuyến nghị sản phẩm; dữ liệu bệnh nhân có thể giúp phát hiện sớm nguy cơ bệnh lý...

Tuy nhiên, việc khai thác loại dữ liệu này một cách hiệu quả vẫn là một thách thức lớn. Làm thế nào để thu thập, xử lý và lưu trữ được khối lượng dữ liệu khổng lồ đó một cách ổn định và kịp thời? Làm thế nào để chuyển hóa dữ liệu thô thành thông tin có ích phục vụ ra quyết định nhanh chóng, trực quan? Đây chính là bài toán quan trọng đặt ra không chỉ với các đơn vị vận tải, cơ quan quản lý đô thị mà còn có thể mở rộng ra các lĩnh vực khác như thương mại, tài chính, logistics hay quy hoạch đô thị thông minh.

Việc giải quyết bài toán này không chỉ giúp cải thiện hiệu quả quản lý, vận hành mà còn tạo tiền đề cho việc phát triển các hệ thống phân tích dữ liệu lớn có tính ứng dụng cao trong các lĩnh vực khác của nền kinh tế số.

1.2 Mục tiêu và phạm vi đề tài

Hiện nay, có nhiều phương pháp, cách tiếp cận cũng như công cụ hỗ trợ giải quyết vấn đề đã nêu ra ở mục 1.1. Xét về kiến trúc tổng thể, một hệ thống làm việc với dữ liệu lớn có thể triển khai bằng nhiều kiến trúc như Lambda, Kappa, Microservices, Zeta hoặc IoT... [1]. Mỗi kiến trúc có ưu và nhược điểm riêng, phù hợp với các yêu cầu phần cứng, phần mềm khác nhau, được thiết kế để giải quyết những bài toán cụ thể với mục tiêu và loại dữ liệu riêng biệt. Về mặt công nghệ, có hàng trăm công nghệ và công cụ có thể áp dụng cho hệ thống dữ liệu lớn, thường được chia thành năm nhóm chính: Thu thập, lưu trữ, xử lý, phân tích và trực quan hóa dữ liệu, quản lý và điều phối hệ thống.

Tuy nhiên, chính sự đa dạng công nghệ và độ phức tạp trong triển khai hệ thống dữ liệu lớn cũng đặt ra nhiều thách thức, như thiếu tính tích hợp giữa các công cụ, khó đảm bảo độ tin cậy và khả năng khôi phục hệ thống, hạn chế trong khả năng mở rộng, quản lý tài nguyên và giám sát tập trung, đồng thời đòi hỏi kiến thức kỹ thuật sâu rộng về cả dữ liệu lớn lẫn hạ tầng triển khai.

Các hệ thống ngày nay thường triển khai trên môi trường Cloud, nơi mà các dịch vụ được tích hợp tốt trong hệ sinh thái của nhà cung cấp. Mặc dù đây là giải pháp thuận tiện và hiệu quả, nhưng với những hệ thống có yêu cầu cao về bảo mật, kiểm soát dữ liệu – như trong lĩnh vực ngân hàng, tài chính..., hay với những doanh nghiệp muốn toàn quyền kiểm soát, vận hành hệ thống của họ, thì hệ thống vẫn cần triển khai trên môi trường hạ tầng nội bộ (On-Premise). Và việc này thường sẽ dẫn đến vấn đề đã nêu ở trên - thiếu tính tích hợp, thống nhất và rời rạc trong việc triển khai, vận hành, giám sát các thành phần.

Trước vấn đề đó, đề tài này hướng tới việc xây dựng một hệ thống xử lý dữ liệu lớn toàn chỉnh, đầy đủ các mô-đun từ thu thập, xử lý đến lưu trữ, trực quan và quản lý, có khả năng tự vận hành trên môi trường On-Premise nhưng vẫn đảm bảo các tiêu chí về khả năng mở rộng, ổn định và dễ giám sát.

Về phạm vi của đề tài, em sẽ xây dựng một hệ thống với dữ liệu đầu vào là dữ liệu có cấu trúc dạng văn bản và dữ liệu theo sự kiện (Event-based data), được ghi nhận từ các hệ thống như giao dịch, tương tác người dùng hoặc hệ thống cảm biến. Hệ thống hướng tới đáp ứng khả năng xử lý dữ liệu theo lô (Batch) nhằm phục vụ phân tích toàn cục, đồng thời xử lý gần thời gian thực (Near-Realtime) để hỗ trợ phản ứng nhanh và trực quan hóa dữ liệu tức thời. Hệ thống được triển khai và thử nghiệm trong môi trường On-Premise, với khả năng tích hợp các công cụ giám sát, quản lý và điều phối linh hoạt.

1.3 Định hướng giải pháp

Em lựa chọn định hướng triển khai theo kiến trúc Lambda, kết hợp các công nghệ trong hệ sinh thái Big Data như Apache Kafka, Apache Spark, Hadoop HDFS, và triển khai toàn bộ hệ thống trên nền tảng Kubernetes On-Premise. Lý do lựa chọn hướng tiếp cận này là vì kiến trúc Lambda cho phép xử lý dữ liệu hiệu quả theo cả hai hình thức: thời gian thực (streaming) và theo lô (batch), đáp ứng tốt yêu cầu về độ trễ thấp và độ chính xác cao trong phân tích dữ liệu lớn.

Giải pháp được xây dựng bao gồm một pipeline xử lý dữ liệu đầu-cuối, tích hợp đầy đủ các thành phần từ thu thập, phân tích đến lưu trữ và trực quan hóa dữ liệu. Việc triển khai trên Kubernetes giúp đảm bảo khả năng phân phối linh hoạt, mở rộng theo chiều ngang và sát với môi trường sản xuất thực tế.

Ở mô-đun thu thập dữ liệu, em sẽ thực hiện thu thập dữ liệu qua API. Dữ liệu này sẽ được gửi đến hệ thống hàng đợi tin nhắn Kafka, đóng vai trò là trung tâm truyền dữ liệu thời gian thực giữa các thành phần trong hệ thống.

Về mô-đun xử lý, em sử dụng Spark Structured Streaming để xử lý dữ liệu dòng, và Spark SQL để xử lý dữ liệu theo lô định kỳ. Spark cung cấp khả năng xử lý dữ liệu trong bộ nhớ, cho phép tăng tốc độ xử lý so với các hệ thống truyền thống, đồng thời hỗ trợ khả năng mở rộng tốt trên hạ tầng phân tán.

Mô-đun lưu trữ kết hợp giữa Redis (lưu dữ liệu tạm thời, hỗ trợ phân tích real-time), Hadoop HDFS (lưu dữ liệu thô và kết quả batch), và PostgreSQL (lưu dữ liệu đã xử lý phục vụ truy vấn và trực quan hóa).

Phần trực quan hóa được xây dựng bằng công cụ Apache Superset, cho phép tạo các báo cáo và dashboard tương tác trực tiếp với PostgreSQL. Ngoài ra, dữ liệu cũng được hiển thị thông qua một Web Server tùy biến để kiểm tra real-time, tăng tính tương tác và khả năng ứng dụng trong môi trường sản xuất.

Toàn bộ hệ thống được tự động hóa bằng Apache Airflow, đảm nhiệm việc lập lịch, điều phối các tác vụ từ thu thập đến lưu trữ và phân tích dữ liệu. Cuối cùng, hệ thống được triển khai trên môi trường Kubernetes On-premise, cho phép kiểm soát hoàn toàn hệ thống, giám sát hoạt động, sao lưu dữ liệu định kỳ; đảm bảo tính bảo mật, mở rộng và chịu lỗi.

Đóng góp chính của đề án là xây dựng thành công một hệ thống xử lý dữ liệu lớn tích hợp đầy đủ các chức năng từ thu thập đến phân tích và trực quan hóa, hoạt động hiệu quả theo cả hai hình thức batch và streaming. Hệ thống đã được thử nghiệm thực tế trên bộ dữ liệu taxi quy mô lớn, cho thấy khả năng vận hành ổn định, linh hoạt, và có tiềm năng ứng dụng trong các bài toán phân tích dữ liệu lớn ở quy mô

doanh nghiệp.

1.4 Bố cục đề án

Phần còn lại của báo cáo đề án tốt nghiệp được tổ chức như sau.

Chương 2 trình bày khảo sát hiện trạng, phân tích yêu cầu hệ thống bao gồm yêu cầu chức năng, phi chức năng, và tổng quan về hệ thống dữ liệu lớn phục vụ cho bài toán phân tích dữ liệu taxi.

Trong Chương 3, em giới thiệu các công nghệ được sử dụng trong hệ thống, trình bày đặc điểm nổi bật và lý do lựa chọn, làm cơ sở cho quá trình thiết kế và triển khai.

Chương 4 trình bày chi tiết thiết kế hệ thống từ tổng quan đến từng mô-đun cụ thể: thu thập, xử lý, lưu trữ và trực quan hóa dữ liệu. Đồng thời, chương này mô tả cách triển khai các mô-đun lên môi trường Kubernetes và các kết quả thực nghiệm.

Chương 5 trình bày đóng góp của đề án - giải pháp thiết kế môi trường cụm Kubernetes sao cho đảm bảo tính mở rộng, chịu lỗi, hiệu quả và chuyên nghiệp.

Cuối cùng, Chương 6 đưa ra kết luận của đề án, đánh giá ưu điểm, hạn chế của hệ thống và đề xuất định hướng phát triển trong tương lai.

CHƯƠNG 2. KHẢO SÁT VÀ PHÂN TÍCH YÊU CẦU

Trong chương 2, em sẽ tiến hành khảo sát các hệ thống và giải pháp hiện có liên quan đến bài toán xử lý dữ liệu lớn, đồng thời phân tích kiến trúc tổng thể và môi trường triển khai hệ thống. Trên cơ sở đó, chương này sẽ trình bày tổng quan về các chức năng mà hệ thống cần đáp ứng, bao gồm cả yêu cầu chức năng và yêu cầu phi chức năng nhằm đảm bảo hệ thống hoạt động hiệu quả, ổn định và dễ mở rộng.

2.1 Khảo sát hiện trạng

Khi thiết kế và xây dựng một hệ thống xử lý dữ liệu lớn, hai yếu tố then chốt cần được làm rõ ngay từ đầu là kiến trúc hệ thống và môi trường triển khai. Hiện nay có hàng trăm công cụ và nền tảng công nghệ khác nhau, mỗi công cụ phù hợp với những yêu cầu, quy mô, mục tiêu triển khai riêng biệt. Chính vì vậy, việc lựa chọn kiến trúc tổng thể cùng môi trường triển khai phù hợp sẽ ảnh hưởng trực tiếp đến hiệu quả, khả năng mở rộng và tính ứng dụng của hệ thống trong thực tế.

2.1.1 Khảo sát về kiến trúc hệ thống

Trong phần này, em sẽ tiến hành khảo sát một số kiến trúc hệ thống dữ liệu lớn phổ biến trong thực tế, đã và đang áp dụng trong một số doanh nghiệp, đồng thời đánh giá ưu, nhược điểm của từng kiến trúc. Các kiến trúc được đề cập đến bao gồm: Lambda, Kappa, Microservice, Zeta và IoT.

Kiến trúc Lambda [1] là một trong những kiến trúc đầu tiên được triển khai và đã trở thành tiêu chuẩn trong một khoảng thời gian. Đây là kiến trúc hướng tới mục tiêu đạt được độ trễ thấp trong khi duy trì độ chính xác cao nhất của dữ liệu, chia thành ba tầng riêng biệt bao gồm tầng xử lý dữ liệu theo lô (Batch), tầng xử lý dữ liệu thời gian thực (Speed) và tầng phục vụ (Serving). Lambda phù hợp cho những hệ thống cần khả năng xử lý dữ liệu thời gian thực và truy vấn dữ liệu lịch sử. Hạn chế của kiến trúc này là cần phải duy trì sự đồng bộ giữa các tầng xử lý, và cần phải viết hai đoạn mã cho việc xử lý cùng một bộ dữ liệu [2].

Để giải quyết nhược điểm trên của Lambda, kiến trúc Kappa [1] đã ra đời với khả năng duy trì một bộ mã nguồn duy nhất, cho phép đồng thời xử lý dữ liệu thời gian thực và truy vấn dữ liệu lịch sử thông qua cơ chế phát lại (Replay) [2]. Tuy nhiên, đặc điểm này lại khiến Kappa không thể lưu trữ dữ liệu lâu dài. Tức là, dữ liệu chỉ được giữ lại trong một khoảng thời gian đã định sẵn, sau đó sẽ bị loại bỏ khỏi hệ thống. Trên thực tế, kiến trúc Kappa đã được áp dụng cho LinkedIn, nơi tác giả của nó - Jay Kreps - làm việc. Ngoài ra, Kappa còn được áp dụng cho một số hệ thống tập trung vào xử lý dữ liệu thời gian thực như IoT, phân tích dữ liệu mạng

xã hội. . .

Kiến trúc Microservice bao gồm tập hợp của các dịch vụ rời rạc, liên kết lỏng lẻo, giao tiếp với nhau thông qua REST API hoặc các lời gọi từ xa (Remote calls). Mỗi dịch vụ sẽ chạy trên máy chủ riêng, đảm nhiệm một chức năng riêng biệt và chính nó cũng đã là một ứng dụng hoàn chỉnh. Với những đặc điểm này, kiến trúc Microservice được áp dụng cho các hệ thống lớn như Amazon, Netflix và eBay [2]. Tuy nhiên, kiến trúc Microservice cũng có những thách thức đáng kể như phức tạp trong việc xây dựng cơ chế giao tiếp giữa các thành phần sao cho đảm bảo tính bảo mật, yêu cầu về hạ tầng lớn. . .

Tiếp theo là kiến trúc Zeta, kiến trúc này đề xuất một cách tiếp cận mới, đó là tích hợp trực tiếp vào kiến trúc nghiệp vụ (Business Architecture). Zeta cung cấp các container – là các môi trường biệt lập – nơi phần mềm có thể được triển khai và tương tác với nhau mà không bị ràng buộc bởi sự khác biệt nền tảng. Nhờ đó, nhiều loại ứng dụng khác nhau có thể cùng được triển khai và vận hành trên kiến trúc Zeta. Kiến trúc Zeta đã được áp dụng cho hệ thống phân bổ chỗ đỗ xe động, hay Gmail của Google [2]. Hạn chế của kiến trúc này là sự phức tạp trong triển khai, khó áp dụng cho các hệ thống đơn giản hoặc quy mô nhỏ.

Cuối cùng là kiến trúc IoT. Như tên gọi, kiến trúc này được đề xuất như là chuẩn thống nhất để áp dụng cho tất cả các hệ thống IoT. Tuy nhiên, kiến trúc này chưa được áp dụng trong thực tế một cách rộng rãi, chủ yếu đang trong quá trình thử nghiệm và kiểm chứng.

2.1.2 Khảo sát về môi trường triển khai

Trong việc xây dựng các hệ thống nói chung và hệ thống dữ liệu lớn nói riêng, việc lựa chọn môi trường triển khai cũng đóng một vai trò quan trọng không kém so với lựa chọn kiến trúc, công nghệ sử dụng. Ba môi trường triển khai phổ biến nhất hiện nay là Cloud (điện toán đám mây), On-Premise (tự vận hành tại chỗ) hoặc Hybrid (kết hợp cả hai). Một số nhà cung cấp dịch vụ Cloud có thể kể đến như Amazon Web Services (AWS), Google Cloud Platform và Microsoft Azure [3]. Việc lựa chọn môi trường nào phụ thuộc vào nhiều yếu tố khác nhau như chi phí, bảo mật dữ liệu, khả năng mở rộng, hiệu suất, nhân lực. . . Bảng 2.1 dưới đây là những đặc điểm nổi bật của ba môi trường triển khai trên.

Bảng 2.1: Các mô hình triển khai hệ thống dữ liệu lớn [4]

	On-Premise	Cloud	Hybrid
Mức độ kiểm soát	Kiểm soát hoàn toàn hạ tầng và dữ liệu nội bộ	Kiểm soát gián tiếp qua SLA, ít khả năng can thiệp trực tiếp	Kiểm soát cân bằng: chủ động với phần tại chỗ, kết hợp cloud linh hoạt
Chi phí	Chi phí đầu tư ban đầu cao, vận hành ổn định	Đầu tư thấp, trả theo mức dùng nhưng khó dự đoán	Cân bằng, nhưng có thể cao vì quản lý song song hai nền tảng
Khả năng mở rộng	Bị giới hạn bởi hạ tầng vật lý, cần đầu tư khi mở rộng	Rất linh hoạt, điều chỉnh nhanh theo nhu cầu	Có thể mở rộng, nhưng bị giới hạn bởi phần hạ tầng nội bộ
Bảo mật	Cao nếu có biện pháp phù hợp	Cao nhưng phụ thuộc vào bảo mật của nhà cung cấp	Cao, cần tích hợp cẩn thận để đảm bảo an toàn
Chủ quyền dữ liệu	Dữ liệu nằm toàn bộ trong hệ thống nội bộ	Dữ liệu ở trung tâm dữ liệu của nhà cung cấp, có thể toàn cầu	Dữ liệu có thể phân tách: phần nhạy cảm lưu tại chỗ
Hiệu năng	Tối ưu tốt cho người dùng nội bộ, độ trễ thấp	Có thể gặp độ trễ, nhưng cải thiện bằng cách chọn trung tâm dữ liệu gần	Hiệu năng linh hoạt, chọn nơi xử lý phù hợp
Bảo trì	Cần đội ngũ nội bộ thực hiện bảo trì	Do nhà cung cấp dịch vụ quản lý	Cần duy trì bảo trì cả hai phía
Đổi mới công nghệ	Bị giới hạn bởi nguồn lực và chuyên môn nội bộ	Dễ tiếp cận công nghệ mới nhờ nhà cung cấp liên tục cập nhật	Có thể ứng dụng công nghệ mới cho phần cloud

Từ khảo sát các kiến trúc hệ thống dữ liệu lớn, có thể thấy rằng kiến trúc Lambda là lựa chọn phù hợp khi cần xử lý đồng thời dữ liệu theo lô và dữ liệu thời gian thực, đảm bảo độ chính xác và khả năng mở rộng. Dù cần duy trì hai luồng xử lý riêng biệt, kiến trúc này cho phép tách biệt rõ ràng vai trò, dễ giám sát và tối ưu hiệu suất. So với Kappa (không lưu dữ liệu lâu dài) hay Microservice, Zeta (yêu cầu hạ tầng phức tạp), Lambda có tính ổn định và khả thi cao trong triển khai nội bộ.

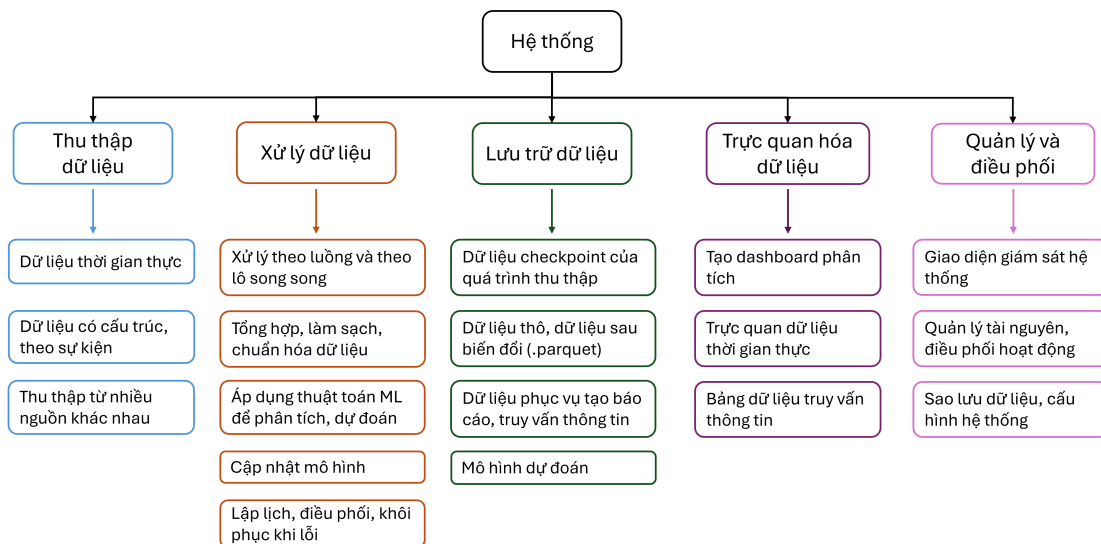
Về môi trường triển khai, mặc dù Cloud cung cấp khả năng mở rộng linh hoạt, nhưng lại đặt ra thách thức về kiểm soát dữ liệu và bảo mật – đặc biệt với các tổ chức yêu cầu dữ liệu không được ra ngoài. Do đó, môi trường On-Premise được lựa chọn trong đề tài này nhằm đảm bảo quyền kiểm soát toàn diện, dễ thử nghiệm, triển khai và bảo trì tại chỗ.

Ngoài ra, khảo sát các hệ thống dữ liệu lớn hiện nay cho thấy xu hướng hiện tại là triển khai trên Cloud, nơi cung cấp sẵn các dịch vụ tích hợp tốt với nhau. Tuy nhiên, với những hệ thống cần triển khai trên môi trường On-Premise sẽ gặp khó khăn trong việc kết hợp các thành phần lại với nhau, đồng thời phải có khả năng thay đổi linh hoạt các công nghệ mới. Việc xây dựng một hệ thống tích hợp đầy đủ các mô-đun chức năng (thu thập, xử lý, lưu trữ, trực quan hóa và giám sát điều phối) là cần thiết để đảm bảo tính đồng bộ, dễ quản lý và sẵn sàng mở rộng trong tương lai.

2.2 Tổng quan chức năng

2.2.1 Yêu cầu chức năng

Hệ thống được xây dựng nhằm thực hiện toàn bộ chu trình xử lý dữ liệu lớn theo kiến trúc Lambda một cách tự động và có thể mở rộng. Các mô-đun tương ứng với các nhóm chức năng bao gồm: thu thập dữ liệu, xử lý dữ liệu, lưu trữ dữ liệu, trực quan hóa dữ liệu và quản lý điều phối hệ thống. Cụ thể các chức năng của từng mô-đun được trình bày trong hình 2.1



Hình 2.1: Yêu cầu chức năng của hệ thống

Mô-đun thu thập dữ liệu cần có khả năng tiếp nhận dữ liệu thời gian thực với định dạng có cấu trúc, theo dạng sự kiện (event-based). Hệ thống hỗ trợ thu thập dữ liệu từ nhiều nguồn khác nhau như API, tệp tin, hệ thống giao dịch hoặc cảm

biến IoT, đồng thời cần có khả năng mở rộng linh hoạt trong tương lai.

Mô-đun xử lý dữ liệu hỗ trợ xử lý đồng thời cả dữ liệu theo luồng (Streaming) và dữ liệu theo lô (Batch). Quá trình xử lý bao gồm các bước tổng hợp, làm sạch và chuẩn hóa dữ liệu, đồng thời tích hợp khả năng áp dụng các thuật toán học máy (machine learning) để phân tích và dự đoán. Ngoài ra, hệ thống cần hỗ trợ cập nhật mô hình định kỳ, toàn bộ quy trình xử lý được thực thi tự động và thực hiện lại khi xảy ra lỗi.

Mô-đun lưu trữ dữ liệu có trách nhiệm lưu lại toàn bộ dữ liệu đầu vào, trung gian và đầu ra. Cụ thể, hệ thống cần lưu trữ dữ liệu checkpoint phục vụ phục hồi luồng dữ liệu, dữ liệu thô và dữ liệu sau biến đổi (dạng Parquet), dữ liệu phục vụ cho truy vấn phân tích cũng như mô hình dự đoán được huấn luyện. Mô-đun này cũng lưu trữ dữ liệu phục vụ truy vấn qua API, kết nối đến dịch vụ Web.

Mô-đun trực quan hóa dữ liệu cần cung cấp các công cụ, giao diện để người dùng có thể tạo dashboard phân tích, biểu đồ dữ liệu và truy vấn thông tin từ các bảng dữ liệu. Ngoài ra, hệ thống cần hỗ trợ hiển thị dữ liệu thời gian thực để theo dõi diễn biến theo thời gian.

Cuối cùng, mô-đun quản lý và điều phối đóng vai trò kiểm soát toàn bộ hoạt động của hệ thống. Các yêu cầu chức năng bao gồm cung cấp giao diện giám sát trạng thái hoạt động, quản lý tài nguyên tính toán và luồng dữ liệu, đồng thời hỗ trợ sao lưu dữ liệu, cấu hình hệ thống nhằm đảm bảo tính ổn định và an toàn.

2.2.2 Yêu cầu phi chức năng

Ngoài các yêu cầu về chức năng kể trên, hệ thống cần bảo đảm một số yêu cầu phi chức năng nhằm đảm bảo hoạt động hiệu quả, tin cậy và duy trì ổn định trong thời gian dài.

Thứ nhất, hệ thống cần đạt hiệu suất cao trong việc xử lý lượng lớn dữ liệu liên tục. Hệ thống phải tối ưu về tốc độ xử lý, độ trễ thấp nhưng vẫn đảm bảo độ chính xác trong kết quả phân tích. Ngoài ra, hệ thống cần có khả năng tích hợp tốt với những phần mềm, dịch vụ khác.

Thứ hai, hệ thống cần có khả năng mở rộng linh hoạt để dễ dàng thích nghi với sự gia tăng của khối lượng dữ liệu và số lượng người dùng. Điều này đòi hỏi hạ tầng và kiến trúc hệ thống phải hỗ trợ mở rộng theo chiều ngang, phân tán tốt trên các máy chủ xử lý.

Thứ ba, hệ thống phải đảm bảo tính sẵn sàng cao, độ tin cậy và khả năng chịu lỗi. Dữ liệu cần được lưu trữ ra một nơi riêng biệt, đảm bảo các dịch vụ có thể truy cập dễ dàng. Các cơ chế sao lưu dữ liệu định kỳ, duy trì hoạt động khi cập nhật,

tái khởi động tác vụ khi gặp lỗi và phân phối tải hợp lý cần được tích hợp để giảm thiểu rủi ro gián đoạn trong quá trình vận hành.

Thứ tư, hệ thống cần cung cấp một giao diện quản lý trực quan, cho phép người quản trị dễ dàng theo dõi trạng thái các mô-đun, kiểm tra log và giám sát tài nguyên. Giao diện này cũng đóng vai trò là công cụ quan trọng để đánh giá hiệu suất hệ thống và hỗ trợ quyết định kỹ thuật kịp thời.

CHƯƠNG 3. CÔNG NGHỆ SỬ DỤNG

3.1 Công nghệ lập lịch, điều phối - Apache Airflow

Apache Airflow [5] là một nền tảng mã nguồn mở dùng để lập lịch, điều phối và theo dõi trình tự xử lý dữ liệu. Được phát triển bởi Airbnb và hiện trực thuộc Apache Software Foundation, Airflow cho phép người dùng định nghĩa các luồng công việc (DAG – Directed Acyclic Graph) bằng ngôn ngữ Python, từ đó giúp tự động hóa các tác vụ ETL (Extract - Transform - Load), kiểm tra dữ liệu, huấn luyện mô hình, hoặc cập nhật dashboard.

Một DAG trong Airflow bao gồm nhiều tác vụ (task) có quan hệ phụ thuộc rõ ràng về thứ tự và logic. Airflow hỗ trợ nhiều loại tác vụ như chạy script Python, shell, Spark job, gửi email, hoặc thao tác với API bên ngoài. Bên cạnh đó, Airflow cung cấp khả năng lập lịch định kỳ (cron-like), thực thi lại tác vụ khi lỗi, gửi cảnh báo khi có sự cố và ghi log chi tiết cho từng bước xử lý.

Điểm mạnh nổi bật của Airflow là khả năng mở rộng và tích hợp tốt với hệ sinh thái Big Data: người dùng có thể sử dụng các Operator (thành phần thực thi tác vụ) có sẵn như BashOperator, PythonOperator, SparkSubmitOperator, DockerOperator,... hoặc tự xây dựng Operator riêng để phù hợp với hệ thống của mình. Giao diện web của Airflow cho phép quản lý, theo dõi trạng thái task theo thời gian thực và dễ dàng điều chỉnh khi cần.

Trong các hệ thống dữ liệu lớn theo kiến trúc Lambda, Airflow thường được dùng để lập lịch xử lý batch định kỳ, kích hoạt Spark job, đồng bộ dữ liệu giữa các hệ lưu trữ hoặc kiểm tra chất lượng dữ liệu trước khi xuất bản. Khả năng ghi nhớ trạng thái (Stateful Execution) và tương thích với Kubernetes (xem phần 3.7) khiến Airflow trở thành lựa chọn hàng đầu trong các hệ thống dữ liệu hiện đại, yêu cầu tự động hóa và khả năng giám sát cao.

3.2 Công nghệ thu thập dữ liệu - Apache Kafka

Apache Kafka [6] là một nền tảng truyền tải dữ liệu phân tán mã nguồn mở theo mô hình publish-subscribe, được phát triển ban đầu bởi LinkedIn và sau đó đóng góp cho Apache Software Foundation. Kafka được thiết kế với khả năng xử lý dữ liệu theo thời gian thực với độ trễ thấp và khả năng mở rộng cao. Kafka hoạt động như một hệ thống trung gian (middleware) cho phép các ứng dụng publish (gửi), subscribe (nhận), lưu trữ và xử lý luồng sự kiện (event streams) một cách liên tục và tin cậy.

Kafka hoạt động dựa trên khái niệm topic – nơi lưu trữ các luồng dữ liệu. Các

producer ghi dữ liệu vào topic, trong khi các consumer đọc dữ liệu từ đó. Một topic được chia thành nhiều partition, và mỗi partition có thể được sao lưu để đảm bảo độ tin cậy. Dữ liệu trong Kafka được lưu theo thứ tự ghi, không bị thay đổi và có thể truy xuất lại nhiều lần. Kafka có thể mở rộng bằng cách thêm các broker mới vào cụm. Các broker cùng nhau tạo thành một cluster chịu trách nhiệm lưu trữ và quản lý dữ liệu theo phân vùng. Zookeeper được sử dụng để quản lý trạng thái cluster và đồng bộ metadata, mặc dù phiên bản Kafka mới đã giảm phụ thuộc vào Zookeeper. Apache Kafka tích hợp rất tốt với Spark Structured Streaming, mang lại khả năng thu thập và xử lý dữ liệu thời gian thực một cách hiệu quả.

3.3 Công nghệ xử lý dữ liệu - Apache Spark

Apache Spark [7] là nền tảng xử lý dữ liệu phân tán mã nguồn mở, nổi bật với khả năng xử lý song song trong bộ nhớ (in-memory). Spark cung cấp các API cấp cao trong các ngôn ngữ như Java, Scala, Python và R, cùng với một bộ máy thực thi được tối ưu hóa hỗ trợ đồ thị thực thi tổng quát (general execution graphs). Ngoài ra, Spark còn hỗ trợ một tập hợp phong phú các công cụ cấp cao, bao gồm: Spark SQL, Pandas API, MLlib, GraphX, Spark Structured Streaming.

3.3.1 Spark SQL

Spark SQL là thành phần xử lý dữ liệu có cấu trúc của Apache Spark, cho phép người dùng sử dụng ngôn ngữ SQL để truy vấn và phân tích dữ liệu. Nó hỗ trợ tích hợp với các định dạng dữ liệu phổ biến như Parquet, Avro, JSON và Hive, đồng thời tận dụng khả năng tối ưu hóa của Catalyst Optimizer để tăng hiệu suất xử lý.

3.3.2 Spark MLlib

MLlib là thư viện học máy của Spark, cung cấp các thuật toán học máy phổ biến như phân cụm (clustering), phân loại (classification), hồi quy (regression) và các công cụ tiền xử lý dữ liệu. MLlib cho phép triển khai và huấn luyện các mô hình học máy trên tập dữ liệu lớn một cách hiệu quả, đồng thời tận dụng khả năng tính toán phân tán của Spark.

3.3.3 Spark Structured Streaming

Spark Structured Streaming là mô-đun xử lý dữ liệu theo luồng (streaming) của Spark, được xây dựng dựa trên Spark SQL engine. Nó vận hành theo mô hình micro-batch, xử lý dữ liệu theo từng lô nhỏ liên tục, và cho phép sử dụng lại các phép biến đổi (transformation) giống như trong Spark SQL. Nhờ đó, Structured Streaming đơn giản hóa việc triển khai các kiến trúc như Lambda, vừa đáp ứng yêu cầu xử lý thời gian thực, vừa đảm bảo tính ổn định và mở rộng.

3.4 Công nghệ lưu trữ dữ liệu

3.4.1 Hadoop HDFS

Hadoop Distributed File System (HDFS) [8] là hệ thống tệp phân tán được phát triển trong khuôn khổ của Apache Hadoop, nhằm lưu trữ dữ liệu khối lượng lớn trên nhiều máy chủ một cách an toàn và hiệu quả. HDFS tuân theo mô hình master-slave, trong đó NameNode chịu trách nhiệm quản lý metadata (thông tin về cấu trúc file và vị trí block), còn các DataNode lưu trữ trực tiếp dữ liệu người dùng. Mỗi file trong HDFS được chia thành các block có kích thước mặc định (thường là 128MB hoặc 256MB), và mỗi block được nhân bản trên nhiều node để đảm bảo tính sẵn sàng và độ tin cậy cao.

Một điểm mạnh quan trọng của HDFS là khả năng xử lý song song và phân tán: các tác vụ đọc/ghi có thể thực hiện đồng thời trên nhiều DataNode. Bên cạnh đó, HDFS được thiết kế tối ưu cho các ứng dụng đọc tuần tự dữ liệu lớn, phù hợp với các hệ thống xử lý theo lô như Spark hoặc MapReduce. Tuy nhiên, HDFS không phù hợp cho các ứng dụng yêu cầu đọc-ngẫu nhiên hoặc ghi dữ liệu thường xuyên với độ trễ thấp.

HDFS thường đóng vai trò là nơi lưu trữ trung tâm trong các hệ thống dữ liệu lớn, nơi dữ liệu thô (raw data) được lưu giữ lâu dài và phục vụ cho các tác vụ xử lý batch. Khi kết hợp với các hệ thống như YARN, Hive, hoặc Spark, HDFS cung cấp một tầng lưu trữ mạnh mẽ, dễ mở rộng và có khả năng tích hợp cao trong các pipeline xử lý dữ liệu hiện đại.

3.4.2 Redis

Redis [9] là một hệ quản trị cơ sở dữ liệu dạng key-value chạy trong bộ nhớ (in-memory), nổi bật với tốc độ truy xuất dữ liệu rất cao và được thiết kế tối ưu cho các ứng dụng thời gian thực. Được phát triển từ năm 2009, Redis hỗ trợ nhiều kiểu dữ liệu như string, list, set, hash và sorted set. Đây là điểm khác biệt lớn so với các hệ thống cache truyền thống chỉ hỗ trợ dạng khóa-giá trị đơn giản.

Redis thường được sử dụng để làm cache, hàng đợi tác vụ (task queue), bộ đếm thời gian thực hoặc lưu trữ tạm thời cho dữ liệu trung gian. Nhờ khả năng đọc/ghi cực nhanh và hỗ trợ các thao tác nguyên tử (atomic operations), Redis là lựa chọn lý tưởng trong các hệ thống cần phản hồi nhanh hoặc xử lý song song cao.

Redis hỗ trợ cấu hình phân cụm (cluster) và nhân bản (replication), cùng với tính năng bền vững (persistence) thông qua cơ chế snapshot hoặc ghi log theo thời gian thực (AOF – Append Only File). Ngoài ra, Redis có thể được tích hợp với các hệ thống như Spark Streaming, Kafka hoặc Flask để xây dựng các kiến trúc

microservice hiệu quả.

Trong hệ thống dữ liệu lớn, Redis có thể đóng vai trò như bộ nhớ đệm (buffer) giữa các bước xử lý, nơi lưu trữ tạm thời kết quả trung gian hoặc dữ liệu stream. Điều này giúp giảm tải cho các hệ thống lưu trữ chính và tăng hiệu suất tổng thể.

3.4.3 PostgreSQL

PostgreSQL [10] là hệ quản trị cơ sở dữ liệu quan hệ mã nguồn mở, được phát triển từ năm 1986 tại Đại học California tại Berkeley, và đã trở thành một trong những hệ thống cơ sở dữ liệu phổ biến và mạnh mẽ nhất hiện nay. PostgreSQL hỗ trợ đầy đủ chuẩn SQL và mở rộng thêm các tính năng nâng cao như xử lý JSON, indexing đa dạng (B-tree, GIN, GiST), stored procedures, và cơ chế giao dịch ACID.

Một điểm nổi bật của PostgreSQL là khả năng mở rộng mạnh mẽ và tính linh hoạt cao. Người dùng có thể định nghĩa kiểu dữ liệu riêng, hàm mở rộng, hoặc module xử lý song song. PostgreSQL cũng hỗ trợ đồng thời xử lý nhiều truy vấn phức tạp, phân vùng dữ liệu và thực thi các phép toán phân tích.

Trong các hệ thống phân tích dữ liệu hiện đại, PostgreSQL thường đóng vai trò là kho lưu trữ kết quả (analytical sink), nơi tổng hợp, lưu trữ và cung cấp dữ liệu cho các công cụ trực quan hóa như Superset, Metabase hoặc Tableau. Với khả năng tích hợp tốt, bảo mật cao và cộng đồng hỗ trợ mạnh, PostgreSQL là lựa chọn hàng đầu trong nhiều hệ thống ETL và BI hiện đại.

Để có thể thực hiện tốt chức năng của mô-đun lưu trữ dữ liệu, cần kết hợp khả năng lưu trữ của HDFS, Redis và PostgreSQL. Cụ thể, HDFS được thiết kế để lưu trữ dữ liệu khối lượng lớn trong thời gian dài và tối ưu cho xử lý theo lô (batch), phù hợp cho tầng lưu trữ dữ liệu gốc, dữ liệu bán xử lý và dữ liệu các mô hình học máy. Trong khi đó, Redis với tốc độ truy xuất nhanh và khả năng lưu trữ tạm thời trong bộ nhớ là lựa chọn lý tưởng cho các tác vụ đệm (buffering), lưu kết quả trung gian hoặc phục vụ dữ liệu thời gian thực. Cuối cùng, PostgreSQL- với khả năng tích hợp tốt với các công cụ trực quan hóa và Web Server - đóng vai trò là nơi lưu trữ kết quả phân tích cuối cùng.

3.5 Công nghệ trực quan hóa dữ liệu

3.5.1 Superset

Apache Superset [11] là một nền tảng khám phá, trực quan hóa dữ liệu mã nguồn mở mạnh mẽ được phát triển bởi Airbnb và hiện thuộc dự án của Apache Software Foundation. Superset cho phép người dùng tạo ra các bảng điều khiển (Dashboard), biểu đồ, và báo cáo tương tác mà không cần viết nhiều mã, nhờ giao diện đồ họa

thân thiện và khả năng kết nối với nhiều hệ quản trị cơ sở dữ liệu như PostgreSQL, MySQL, ClickHouse, Snowflake và nhiều hệ thống khác.

Superset hỗ trợ truy vấn SQL trực tiếp, cho phép người dùng nâng cao khả năng phân tích dữ liệu bằng các biểu thức tùy chỉnh. Superset cũng cung cấp chế độ lọc động (filter), phân quyền người dùng chi tiết (RBAC – role-based access control) và cơ chế lưu cache để tăng hiệu năng hiển thị. Người dùng có thể xây dựng nhiều loại biểu đồ như line, bar, pie, sunburst, heatmap hoặc bản đồ địa lý với khả năng cấu hình cao.

Bên cạnh đó, Superset còn hỗ trợ xuất dữ liệu, cấu hình metadata source, và hiển thị log truy vấn để debug. Đây là lựa chọn thay thế nhẹ hơn so với các giải pháp BI thương mại như Tableau hay Power BI, phù hợp với hệ thống dữ liệu lớn mã nguồn mở.

3.5.2 Spring Boot và ReactJS

Trong dự án này, em sử dụng Spring Boot và ReactJS để xây dựng một trang web trực quan hóa các kết quả phân tích, hướng đến sự mở rộng của đồ án (Có thể tích hợp với các dịch vụ khác như Web...)

Spring Boot [12] là một framework Java mã nguồn mở giúp đơn giản hóa quá trình phát triển ứng dụng backend. Nó cung cấp cấu hình mặc định, tích hợp sẵn các module như REST API, bảo mật (Spring Security), truy cập cơ sở dữ liệu (JPA/Hibernate) và dễ dàng triển khai với Docker hoặc Kubernetes.

ReactJS [13] là thư viện JavaScript do Facebook phát triển, chuyên dùng để xây dựng giao diện người dùng tương tác và linh hoạt. React cho phép chia nhỏ giao diện thành các component, giúp dễ tái sử dụng và duy trì. React hỗ trợ cơ chế state management, routing, và tích hợp dễ dàng với các backend như Spring Boot thông qua API REST hoặc WebSocket.

Sự kết hợp giữa Spring Boot và ReactJS cho phép xây dựng các ứng dụng full-stack mạnh mẽ: backend xử lý logic và quản lý dữ liệu, trong khi frontend đảm nhận việc hiển thị, tương tác và cập nhật thời gian thực. Mô hình này thường được dùng trong hệ thống phân tích dữ liệu để hiển thị kết quả đầu ra, biểu đồ động hoặc dashboard tương tác.

3.6 Công nghệ giám sát hệ thống

3.6.1 Prometheus

Prometheus [14] là hệ thống giám sát và cảnh báo mã nguồn mở được phát triển bởi SoundCloud và hiện là một dự án chính thức trong CNCF (Cloud Native Computing Foundation). Prometheus được thiết kế đặc biệt để thu thập và lưu trữ

dữ liệu dạng chuỗi thời gian (time-series) từ các dịch vụ và hệ thống phân tán. Cơ chế thu thập của Prometheus là pull-based, nghĩa là nó sẽ định kỳ lấy dữ liệu từ các endpoint có định dạng chuẩn.

Prometheus lưu trữ dữ liệu cục bộ trên đĩa bằng định dạng riêng, hỗ trợ truy vấn bằng ngôn ngữ PromQL – một ngôn ngữ mạnh mẽ cho phép tính toán, lọc, và phân tích số liệu phức tạp. Hệ thống cũng hỗ trợ cảnh báo thông qua Alertmanager, cho phép gửi thông báo qua email, Slack, Telegram hoặc tích hợp các webhook tùy chỉnh.

Prometheus đặc biệt hiệu quả trong giám sát hệ thống container và các dịch vụ microservice. Nó tích hợp tốt với Kubernetes, Grafana (xem phần 3.7 và 3.6.2) và các exporter phổ biến như Node Exporter, cAdvisor, và JMX Exporter. Nhờ khả năng lưu trữ dữ liệu tạm thời, không phụ thuộc vào cơ sở dữ liệu ngoài, Prometheus dễ dàng triển khai, mở rộng và rất phù hợp với môi trường yêu cầu giám sát thời gian thực.

3.6.2 Grafana

Grafana [15] là công cụ trực quan hóa dữ liệu mạnh mẽ, được sử dụng rộng rãi để xây dựng các dashboard giám sát, biểu đồ thời gian thực và phân tích số liệu từ nhiều nguồn dữ liệu khác nhau. Grafana hỗ trợ kết nối tới hơn 30 loại nguồn dữ liệu, bao gồm Prometheus, InfluxDB, MySQL, PostgreSQL, Elasticsearch, và cả các hệ thống qua API.

Điểm mạnh của Grafana là khả năng tạo dashboard động, tùy chỉnh cao, hỗ trợ biểu đồ dạng line, bar, gauge, heatmap, và bảng số liệu. Người dùng có thể xây dựng các biểu đồ trực quan chỉ với thao tác kéo-thả và lọc theo thời gian thực. Grafana cũng hỗ trợ cơ chế cảnh báo trực tiếp trên dashboard, tích hợp xác thực người dùng và phân quyền truy cập.

Grafana thường được triển khai song song với Prometheus để theo dõi hiệu năng hệ thống, trạng thái dịch vụ, và độ trễ mạng. Trong hệ thống dữ liệu lớn, Grafana đóng vai trò là công cụ trung tâm để giám sát cụm Kubernetes thông qua các exporter. Với giao diện hiện đại, dễ sử dụng và cộng đồng hỗ trợ rộng lớn, Grafana là lựa chọn hàng đầu cho việc giám sát hệ thống thời gian thực.

3.7 Công nghệ triển khai, điều phối - Kubernetes

Kubernetes (K8s) [16] là nền tảng mã nguồn mở dùng để tự động hóa việc triển khai, mở rộng và quản lý các ứng dụng container do Google phát triển, hiện được duy trì bởi CNCF (Cloud Native Computing Foundation). Kubernetes cho phép hệ thống mở rộng linh hoạt, tự phục hồi (self-healing), cân bằng tải, và cập nhật liên

tục mà không làm gián đoạn dịch vụ.

Một cụm K8s bao gồm một Control Plane và một hoặc nhiều Worker node. Thành phần điều khiển trung tâm là Control Plane, nơi quản lý các node, lịch trình triển khai (scheduler), trạng thái của container (kubelet), và API server phục vụ giao tiếp. Các tác vụ tính toán và chạy ứng dụng thực tế được thực hiện bởi các Worker node. Mỗi node trong cụm có thể chạy một hoặc nhiều Pod, là nơi chứa các container ứng dụng. Worker node được quản lý tập trung bởi Control Plane, và chịu trách nhiệm thực thi các lệnh triển khai, cập nhật hoặc phục hồi tài nguyên theo yêu cầu từ hệ thống điều phối.

Trong Kubernetes, có một số thành phần cốt lõi đóng vai trò quan trọng trong triển khai hệ thống. Như đã trình bày ở trên, Pod là đơn vị triển khai nhỏ nhất, đại diện cho một hoặc nhiều container chia sẻ cùng không gian mạng và tài nguyên tạm thời. Deployment được dùng để quản lý các Pod không trạng thái (stateless), đảm bảo số lượng bản sao luôn duy trì ở mức khai báo, hỗ trợ rolling update và tự phục hồi. Với các ứng dụng có trạng thái như cơ sở dữ liệu, Kubernetes sử dụng StatefulSet, giúp đảm bảo mỗi Pod có định danh cố định, thứ tự khởi tạo ổn định và gắn liền với volume riêng. Để kết nối và giao tiếp giữa các Pod hoặc truy cập từ bên ngoài, Kubernetes cung cấp Service, là điểm truy cập ổn định bất chấp việc Pod có thể bị thay thế. Ngoài ra, Ingress được dùng để định tuyến lưu lượng HTTP(S) từ bên ngoài vào bên trong cụm, thường gắn với tên miền và đường dẫn cụ thể. Đối với lưu trữ dữ liệu, Kubernetes cung cấp Persistent Volume (PV) và Persistent Volume Claim (PVC) để tách biệt vòng đời lưu trữ khỏi vòng đời Pod, giúp hệ thống đảm bảo tính bền vững của dữ liệu.

Kubernetes hỗ trợ mô hình quản lý cấu hình khai báo (declarative configuration) thông qua YAML file, giúp quản lý trạng thái hệ thống dễ dàng. Ngoài ra, K8s hỗ trợ các tính năng như Autoscaling, Rolling Updates và Network Policies, phù hợp với yêu cầu triển khai hệ thống phân tán quy mô lớn.

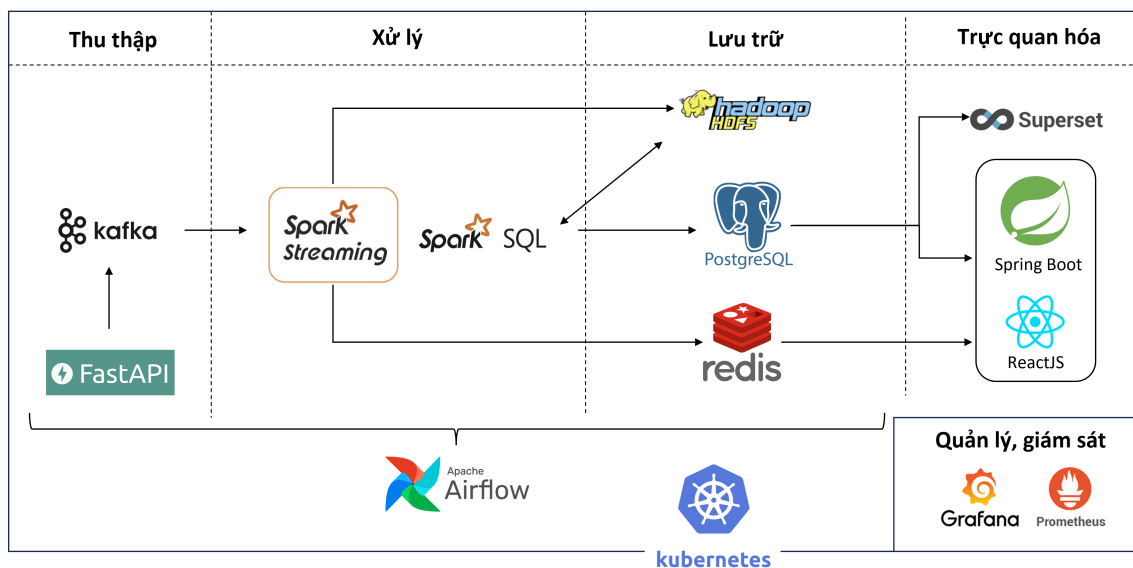
Với kiến trúc linh hoạt và khả năng tích hợp với Docker, Prometheus, Grafana, và Helm, Kubernetes hiện là tiêu chuẩn công nghiệp cho triển khai và điều phối hệ thống hiện đại, đặc biệt là trong các dự án dữ liệu lớn và kiến trúc Microservices.

CHƯƠNG 4. THIẾT KẾ, TRIỂN KHAI VÀ KẾT QUẢ THỰC NGHIỆM

Chương này trình bày chi tiết kiến trúc hệ thống, cách thức triển khai các chức năng nêu ở chương 2 bằng các công cụ đã nêu ở chương 3 cho bài toán về dữ liệu taxi ở New York như một ví dụ thực tiễn. Chất lượng của hệ thống sau khi triển khai cũng sẽ được đánh giá cụ thể.

4.1 Thiết kế kiến trúc

4.1.1 Kiến trúc tổng quan hệ thống



Hình 4.1: Kiến trúc tổng quan của hệ thống

Hình 4.1 mô tả kiến trúc tổng quan của hệ thống, bao gồm các thành phần từ thu thập, xử lý, lưu trữ đến trực quan hóa dữ liệu. Để mô phỏng dữ liệu thời gian thực, luồng dữ liệu bắt đầu từ các thư mục định dạng parquet chứa dữ liệu các chuyến taxi của thành phố New York từ năm 2019 đến 2025. Một ứng dụng FastAPI sẽ giả lập thu thập dữ liệu bằng cách đọc tuần tự các file này theo trường thời gian, sau đó gửi dữ liệu qua API, mô phỏng chính xác luồng dữ liệu thực tế.

Dữ liệu được tiếp nhận bởi Kafka Producer và gửi vào các topic tương ứng. Spark Structured Streaming sẽ đọc dữ liệu từ Kafka, thực hiện các bước tiền xử lý bao gồm: làm sạch, chuẩn hóa, làm giàu, phân tách dữ liệu hợp lệ và lỗi. Dữ liệu hợp lệ được lưu trữ phân tán trên HDFS, đồng thời dữ liệu thời gian thực được lưu tạm trên Redis để phục vụ trực quan hóa.

Redis đóng vai trò như bộ nhớ tạm thời, nơi lưu trữ dữ liệu theo kênh (channel). Một ứng dụng Spring Boot sẽ đăng ký nhận dữ liệu từ kênh này và truyền về frontend ReactJS thông qua WebSocket. Trong khi đó, HDFS không chỉ lưu dữ

liệu thô, mà còn lưu checkpoint của Spark Streaming, các dữ liệu liên quan đến mô hình dự đoán và dữ liệu sau khi biến đổi.

Về luồng xử lý theo lô, Spark SQL sẽ đọc dữ liệu thô trên HDFS, tiến hành biến đổi, xử lý và lưu trữ lên cơ sở dữ liệu PostgreSQL. Dữ liệu này là dữ liệu đã được phân tích, phục vụ truy vấn, tạo báo cáo. PostgreSQL được kết nối trực tiếp với Superset và Spring Boot, phục vụ cho quá trình trực quan hóa dữ liệu.

Toàn bộ quy trình xử lý dữ liệu theo thời gian thực (streaming) và theo lô (batch) đều được điều phối bởi Apache Airflow. Airflow tự động hóa các tác vụ thu thập, xử lý dữ liệu và cập nhật mô hình dự đoán. Ngoài ra, hệ thống còn cung cấp giao diện trực quan hóa cho việc quản lý tài nguyên của các thành phần bằng Prometheus và Grafana. Hệ thống được triển khai toàn bộ trên môi trường Kubernetes On-Premise, cho phép mở rộng linh hoạt, đảm bảo khả năng chịu lỗi, tính sẵn sàng cao và sử dụng tài nguyên tối ưu. Chi tiết về kiến trúc của môi trường Kubernetes sẽ được trình bày ở chương 5.

4.1.2 Mô-đun thu thập dữ liệu

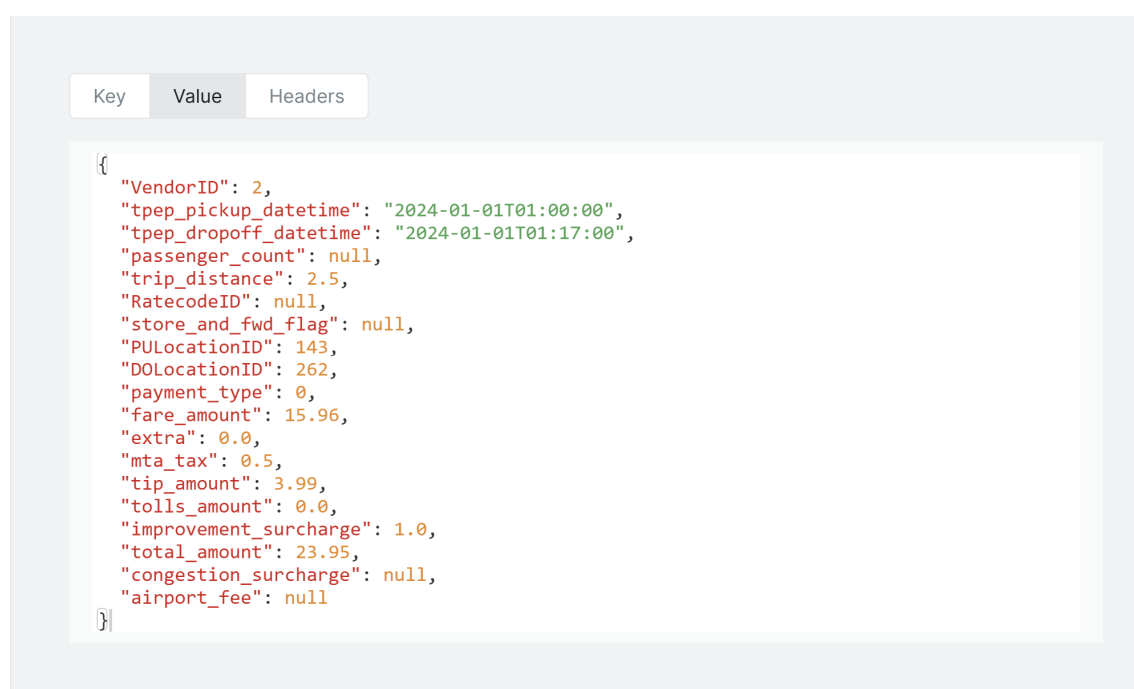
Luồng hoạt động bắt đầu từ việc thu thập dữ liệu. Mô-đun thu thập dữ liệu bao gồm 2 thành phần chính: FastAPI tạo API giả lập thu thập dữ liệu từ các file parquet chứa dữ liệu cho trước và cụm Kafka là nơi trung chuyển dữ liệu từ API đến Kafka Producer để phục vụ cho quá trình biến đổi dữ liệu theo luồng.

Đầu tiên là FastAPI. Để giả lập gửi dữ liệu qua API, em tạo một nơi chứa các thư mục nguồn dữ liệu là /data. Các thư mục ở đây đều có cấu trúc dạng “yellow_tripdata_<year>-<month>.parquet”, với year, month là năm và tháng của dữ liệu. FastAPI sẽ tạo ra một API có đường dẫn /api/taxi_trip, nhận các tham số là type, year, month, day, hour, offset và limit. Từ các tham số này để lấy dữ liệu tương ứng và trả về dưới dạng json. Dữ liệu ban đầu bao gồm các trường được liệt kê trong bảng 4.1 sau:

Bảng 4.1: Thông tin của các trường dữ liệu thu thập

STT	Tên trường	Kiểu dữ liệu	Ý nghĩa
1	VendorID	Int	Mã công ty
2	tpep_pickup_datetime	Timestamp	Thời điểm chuyển bắt đầu
3	tpep_dropoff_datetime	Timestamp	Thời điểm chuyển kết thúc
4	passenger_count	Int	Số lượng hành khách
5	trip_distance	Double	Quãng đường di chuyển (dặm)
6	RatecodeID	Int	Mã loại giá cước
7	store_and_fwd_flag	String	Dữ liệu được gửi ngay lập tức hay không
8	PULocationID	Int	ID của vùng đón khách
9	DOLocationID	Int	ID của vùng trả khách
10	payment_type	Int	Mã phương thức thanh toán (0-6)
11	fare_amount	Double	Cước phí tính theo đồng hồ
12	extra	Double	Phí phụ (giờ cao điểm...)
13	mta_tax	Double	Phí MTA (0.5\$)
14	tip_amount	Double	Tiền tip (chỉ qua thẻ)
15	tolls_amount	Double	Tổng phí cầu đường
16	improvement_surcharge	Double	Phí nâng cấp xe (0.3\$)
17	total_amount	Double	Tổng tiền
18	congestion_surcharge	Double	Phí tắc nghẽn ở Manhattan
19	airport_fee	Double	Phí đón khách ở sân bay

Dữ liệu này sẽ được gửi theo từng bản ghi lên Kafka, với topic ứng với loại taxi. Ví dụ với `type="yellow"` thì topic sẽ là `yellow_trip_data`. Hình 4.2 là ví dụ về một message được gửi lên Kafka, tương ứng với một sự kiện chuyến đi xảy ra. Dữ liệu này sau đó sẽ được đọc bởi Spark Streaming, và thực hiện biến đổi phục vụ lưu trữ, trực quan hóa.

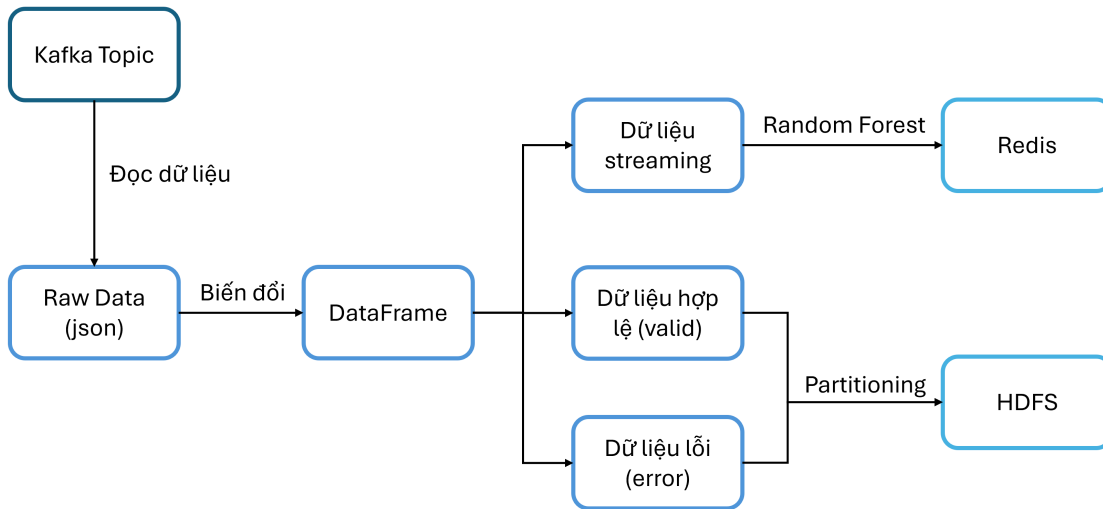


Hình 4.2: Một bản ghi được gửi lên Kafka

4.1.3 Mô-đun xử lý dữ liệu

Với kiến trúc Lambda, hệ thống sẽ gồm hai luồng xử lý dữ liệu là theo luồng (streaming) và theo lô (batch). Ứng với hai luồng xử lý đó em sử dụng Spark Streaming và Spark SQL.

a, Xử lý dữ liệu theo luồng



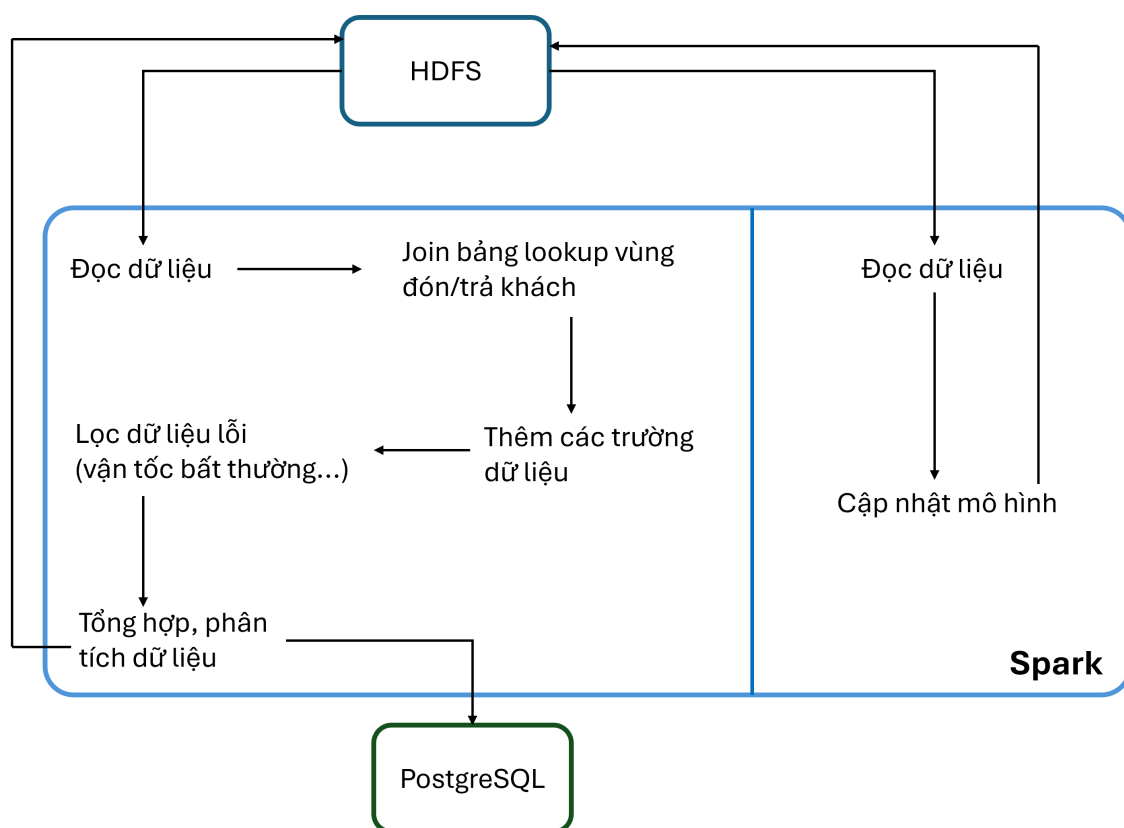
Hình 4.3: Quá trình xử lý dữ liệu theo luồng

Hình 4.3 thể hiện quá trình xử lý dữ liệu theo luồng. Đầu tiên, em định nghĩa lại schema của các trường dữ liệu, đọc dữ liệu từ Kafka và chuyển đổi sang định dạng JSON, sau đó tạo thành DataFrame với schema đã khai báo. Tiếp theo, dữ liệu được làm sạch, bổ sung các trường ngày tháng để phục vụ việc phân vùng khi lưu trữ (partitioning), đồng thời gán giá trị mặc định cho một số trường bị thiếu.

Em chia luồng dữ liệu thành hai phần: `error` (lỗi) và `valid` (hợp lệ), rồi lưu trữ lên HDFS nhằm phục vụ cho việc phân tích lỗi và tiêu thụ dữ liệu sau này. Khi ghi dữ liệu lên HDFS, em sử dụng `checkpointLocation` để lưu lại trạng thái luồng xử lý, đồng thời áp dụng phân vùng theo giờ (hourly partitioning) nhằm tối ưu hóa hiệu suất lưu trữ và truy xuất dữ liệu.

Bên cạnh đó, em còn tích hợp mô hình dự đoán số chuyến đi nhằm phục vụ trực quan hóa dữ liệu gần thời gian thực (near-realtime). Mô hình này được huấn luyện từ dữ liệu lịch sử, sử dụng thuật toán Random Forest [17] để dự đoán số chuyến đi trong một giờ tiếp theo. Trong quá trình xử lý, em áp dụng mô hình ngay khi ghi dữ liệu vào Redis.

Tại Redis, em tạo một khóa `total_trips` để lưu trữ tổng số chuyến đi từ đầu ngày. Mỗi lô dữ liệu (micro-batch) sẽ được xử lý để tạo ra dữ liệu đầu ra dưới dạng JSON, bao gồm các trường “timestamp”, “trip_count”, “predicted_timestamp”, và “predicted_trip_count”. Đây là dữ liệu được đẩy vào kênh `realtime-trip-channel` để phía backend tiếp nhận và sử dụng hiển thị trên biểu đồ thời gian thực ở giao diện người dùng.

b, Xử lý dữ liệu theo lô**Hình 4.4:** Quá trình xử lý dữ liệu theo lô

Hình 4.4 là quá trình xử lý dữ liệu theo lô. Quá trình này bắt đầu từ việc đọc tập dữ liệu trong vòng một tháng, được tạo ra từ quá trình xử lý dữ liệu theo luồng và lưu trữ trước đó trên HDFS. Sau khi đọc dữ liệu, em thực hiện phép nối (join) với bảng thông tin chi tiết về địa điểm, sử dụng hai trường PULocationID và DOLocationID để liên kết với dữ liệu trong tệp taxi_zone_lookup.csv.

Để tối ưu hiệu năng xử lý trong Spark, em áp dụng kỹ thuật Broadcast Join vì bảng địa điểm chỉ chứa 263 bản ghi – rất nhỏ so với tập dữ liệu chuyển đi. Việc broadcast bảng nhỏ này giúp tránh việc shuffle dữ liệu, giảm thời gian xử lý đáng kể khi thực hiện join.

Sau khi hoàn thành bước kết nối, em bổ sung thêm nhiều trường thông tin mới nhằm hỗ trợ cho việc trực quan hóa và phân tích dữ liệu trên Superset, bao gồm các trường ngày, tháng, giờ, nhãn ngày trong tuần, định dạng nhãn tháng (MM/yyyy), loại hình thanh toán (mapping payment_type) từ giá trị số sang tên gọi, quãng đường và thời gian di chuyển được chuẩn hóa sang đơn vị km, phút, và tính thêm tốc độ trung bình của mỗi chuyến đi.

Tiếp đó, em tiến hành làm sạch dữ liệu bằng cách loại bỏ các bản ghi có tốc độ

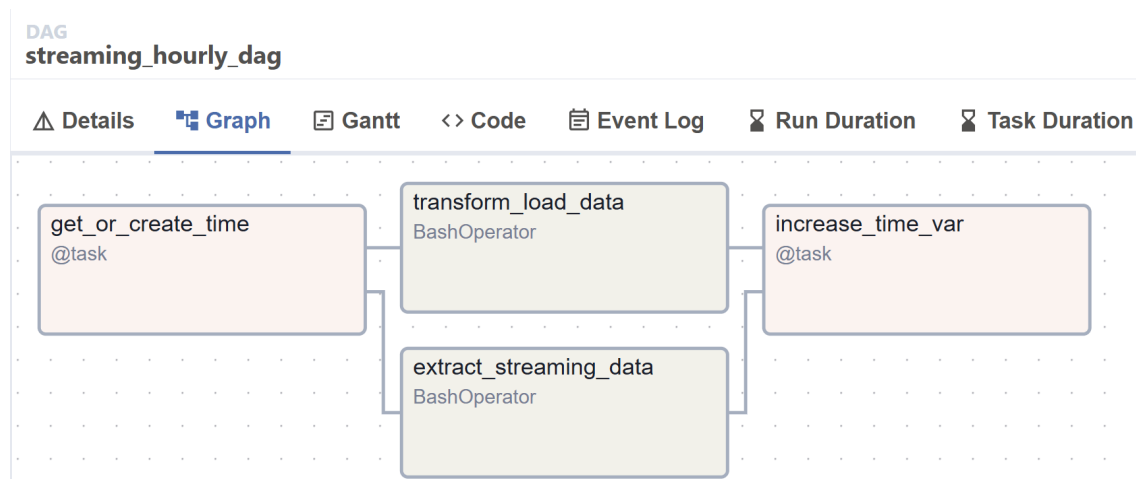
di chuyển bất hợp lý (nhỏ hơn 5 kph (km/giờ) hoặc lớn hơn 120 kph) cũng như các chuyến đi có quãng đường vượt quá 160 km – những trường hợp này được coi là dữ liệu bất thường hoặc lỗi thiết bị.

Bên cạnh việc chuẩn bị dữ liệu cho phân tích chi tiết, em còn thực hiện tổng hợp dữ liệu theo từng tháng để tạo ra tập dữ liệu nhẹ, phục vụ cho truy vấn thống kê từ giao diện web. Dữ liệu này bao gồm các chỉ số như số lượng chuyến đi, doanh thu, quãng đường trung bình, thời gian di chuyển và tốc độ trung bình theo từng tháng, và được lưu trữ đồng thời trên PostgreSQL và HDFS để đáp ứng cả yêu cầu truy vấn nhanh và lưu trữ lâu dài.

Cuối cùng, dữ liệu sau khi xử lý sẽ được kết hợp với các dữ liệu lịch sử để huấn luyện mô hình dự đoán. Việc cập nhật định kỳ mô hình từ dữ liệu mới giúp nâng cao độ chính xác khi dự báo số chuyến đi. Mô hình được huấn luyện sử dụng thuật toán Random Forest, với các đặc trưng đầu vào gồm giờ trong ngày, thứ trong tuần, dấu hiệu ngày cuối tuần và số lượng chuyến đi tại thời điểm hiện tại. Tập dữ liệu được tạo thêm cột `future_trip_count` thông qua phép `lead()` trong Spark, và những bản ghi có đủ dữ liệu đầu ra mới được dùng để huấn luyện. Mô hình sau khi huấn luyện sẽ được lưu dưới dạng pipeline và ghi vào HDFS để phục vụ sử dụng sau này.

4.1.4 Mô-đun lập lịch, tự động

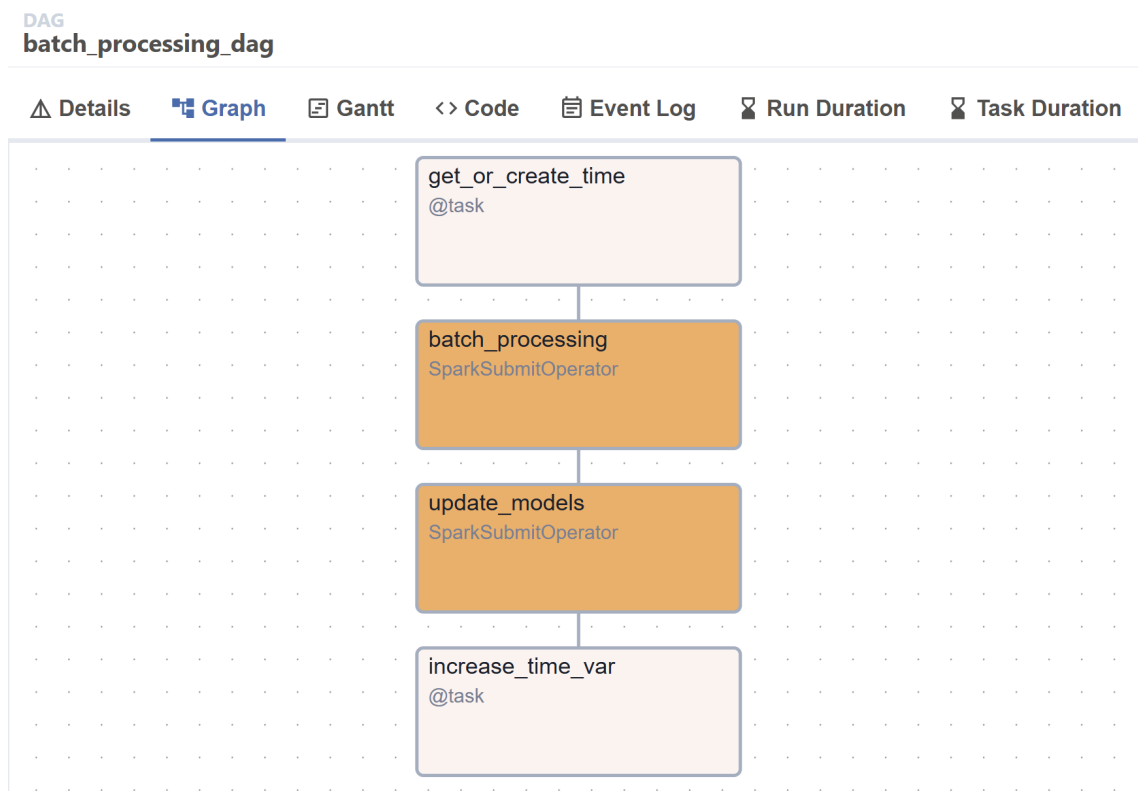
Toàn bộ quá trình thu thập và xử lý dữ liệu trong hệ thống được lập lịch và tự động thực thi thông qua công cụ Apache Airflow. Mỗi luồng ETL (Extract - Transform - Load) được định nghĩa dưới dạng các file DAG (Directed Acyclic Graph) – đồ thị có hướng không chứa chu trình, mô tả tuần tự các bước xử lý dữ liệu trong một quy trình. Trong hệ thống này, em xây dựng hai DAG tương ứng với hai luồng xử lý chính: xử lý theo luồng (streaming) và xử lý theo lô (batch).



Hình 4.5: Đồ thị DAG cho quá trình xử lý streaming

Hình 4.5 minh họa DAG xử lý dữ liệu theo luồng. DAG này gồm hai task chính chạy song song: `extract_streaming_data` và `transform_load_data`, nhằm đảm bảo dữ liệu được tiêu thụ và xử lý đồng thời với quá trình thu thập, từ đó giảm độ trễ trong hiển thị dữ liệu trên giao diện Web. Bên cạnh đó, DAG còn có hai task phụ trợ: `get_or_create_time` và `increase_time_var`. Vì hệ thống sử dụng dữ liệu giả lập near-realtime (không phải dữ liệu thời gian thực), em thiết lập thêm các biến thời gian trong Airflow để mô phỏng luồng dữ liệu như thực tế. Dữ liệu được xử lý theo từng giờ, bắt đầu từ 0h ngày 01/01/2025. DAG được thiết lập với chu kỳ 1 giờ để quá trình diễn ra liên tục và đồng bộ với giả lập thời gian.

Quá trình xử lý theo lô được thiết kế tương tự, với sự khác biệt chủ yếu ở tần suất chạy (mỗi tháng), đường dẫn mã nguồn và biến thời gian riêng biệt. DAG xử lý theo lô được minh họa trong Hình 4.6. Các task trong DAG này được thực hiện tuần tự: từ việc khởi tạo biến thời gian (`get_or_create_time`), xử lý dữ liệu theo lô (`batch_processing`), đến cập nhật lại mô hình dự đoán (`update_models`) và tăng biến đếm thời gian (`increase_time_var`).



Hình 4.6: Đồ thị DAG cho quá trình xử lý theo lô

Airflow cung cấp nhiều loại Operator khác nhau phục vụ cho việc định nghĩa và thực thi các task. Trong DAG xử lý theo luồng, em sử dụng `BashOperator` để chạy mã nguồn tương đương với việc thực thi một lệnh trong terminal. Trong DAG xử lý theo lô (Hình 4.4), em sử dụng `SparkSubmitOperator` để chạy các

script Spark thông qua kết nối tới cụm Spark bên ngoài.

```

1 batch_processing = SparkSubmitOperator(
2     task_id="batch_processing",
3     application="/opt/airflow/code/batch_processing.py"
4     ,
5     name="BatchProcessing",
6     packages="org.postgresql:postgresql:42.7.5",
7     application_args=[
8         "{{ ti.xcom_pull(task_ids='get_or_create_time')
9         ['year'] }}" ,
10        "{{ ti.xcom_pull(task_ids='get_or_create_time')
11        ['month'] }}"
12    ],
13    conn_id="spark_default",
14    verbose=True
15 )
16
17 update_models = SparkSubmitOperator(
18     task_id="update_models",
19     application="/opt/airflow/code/update_predict_model
20     .py",
21     name="UpdateModel",
22     conn_id="spark_default",
23     verbose=True
24 )

```

Khi sử dụng `SparkSubmitOperator`, người dùng cần khai báo một số thuộc tính như: đường dẫn đến file thực thi, các thư viện phụ thuộc (`packages`), các tham số truyền vào (`application_args`), và đặc biệt là `conn_id` – định danh của kết nối Spark được cấu hình trong Airflow. Việc thiết lập kết nối này được thực hiện trong phần `Connections` của Airflow UI, nơi người dùng cần nhập chính xác các thông số như `Host`, `Port`, và `ID` kết nối tương ứng như trong hình 4.7

Edit Connection	
Connection Id *	spark_default
Connection Type *	Spark <small>Connection Type missing? Make sure you've installed the corresponding Airflow Provider Package.</small>
Description	
Host	spark://spark-master-svc.bigdata.svc.cluster.local
Port	7077

Hình 4.7: Kết nối Airflow với Spark để sử dụng SparkSubmitOperator

4.1.5 Mô-đun lưu trữ dữ liệu

a, Redis

Trong kiến trúc hệ thống, Redis là thành phần kết nối với Kafka, Spark Streaming và Web. Redis được sử dụng như một cơ sở dữ liệu lưu trữ dữ liệu tạm thời (in-memory), chuyên dùng để phục vụ các nhu cầu thời gian thực. Vì vậy Redis được sử dụng làm nơi trung gian giữa mô-đun xử lý và trực quan hóa dữ liệu thời gian thực. Dữ liệu sau khi được thu thập bởi Kafka, xử lý bằng Spark Streaming sẽ được lưu xuống Redis. Ở đây em dùng tính năng publish-subscribe, cho phép backend đăng ký lắng nghe kênh realtime-trip-channel và tự động cập nhật khi có dữ liệu mới. Điều này đảm bảo giao diện hiển thị luôn đồng bộ với dữ liệu hiện tại mà không cần phải kiểm tra liên tục trạng thái dữ liệu (polling). Hình 4.8 minh họa dữ liệu thời gian thực trong Redis Channel. Dữ liệu có cấu trúc json, gồm các trường dữ liệu thời gian và dự đoán số chuyến đi. Dữ liệu này sẽ được Web Backend xử lý và Web Frontend sẽ nhận, trực quan hóa dưới dạng biểu đồ đường theo thời gian.

```

1) "message"
2) "realtime-trip-channel"
3) "{
    \"timestamp\": \"2025-01-01 00:01:00\",
    \"trip_count\": 11289,
    \"predicted_timestamp\": \"2025-01-01 01:01:00\",
    \"predicted_trip_count\": 16924
}"
1) "message"
2) "realtime-trip-channel"
3) "{
    \"timestamp\": \"2025-01-01 00:01:04\",
    \"trip_count\": 11291,
    \"predicted_timestamp\": \"2025-01-01 01:01:04\",
    \"predicted_trip_count\": 16926
}"

```

Hình 4.8: Dữ liệu trên kênh realtime-trip-channel

b, HDFS

HDFS là nơi lưu trữ phần lớn dữ liệu, bao gồm cả dữ liệu gốc, dữ liệu biến đổi, dữ liệu checkpoint của quá trình streaming, dữ liệu model dự đoán... Cấu trúc thư mục của HDFS được tổ chức như sau:

```

1 checkpoints/
2   |_ hdfs
3   |_ redis
4
5 raw_data/
6   |_ error
7     |_ year=2023
8       |_ month=1
9     ...
10  |_ valid
11    |_ year=2023
12    |_ year=2024
13    ...
14 processed_data/
15   |_ year=2023
16     |_ month=1
17
18 resources/

```

```

19     |_ taxi_zone_lookup.csv
20
21 models/
22     |_ trip_forecast_model

```

`/checkpoints` là nơi lưu dữ liệu checkpoint cho Spark Streaming để lưu lại trạng thái đọc dữ liệu từ Kafka. `/raw_data` là thư mục lưu toàn bộ dữ liệu thô sau khi đọc từ Kafka, lọc riêng dữ liệu lỗi và hợp lệ, chưa có biến đổi gì. Dữ liệu sau quá trình biến đổi theo lô sẽ được lưu lại ở thư mục `/processed_data` theo từng tháng, và dữ liệu này chính là nguồn để cập nhật các mô hình dự đoán được lưu trong `/models`. Ngoài ra, những dữ liệu khác (như thông tin về các khu vực...) phục vụ cho quá trình biến đổi, làm giàu dữ liệu sẽ được lưu trữ trong `/resources`.

c, PostgreSQL

PostgreSQL là cơ sở dữ liệu SQL, lưu trữ dữ liệu có cấu trúc dưới dạng bảng. Ở đây, em dùng postgresQL làm nơi lưu trữ dữ liệu đã biến đổi phục vụ cho tạo các dashboard báo cáo và dữ liệu phục vụ truy vấn lịch sử trên web. Dữ liệu trên postgresQL được lưu trong database `taxi_trip_db`, bao gồm các bảng `fact_trips`, `analyze_by_time`. Bảng `fact_trips` được biến đổi từ dữ liệu gốc, thêm một số thuộc tính phục vụ cho tạo báo cáo như trong bảng 4.2 sau:

Bảng 4.2: Các trường dữ liệu được thêm trong bảng `fact_trips`

STT	Tên trường	Kiểu dữ liệu	Ý nghĩa
1	hour_label	String	Giờ (dạng HH:mm)
2	payment_type_name	String	Tên phương thức thanh toán
3	trip_distance_km	Double	Quãng đường (km)
4	trip_duration_minutes	Double	Thời gian di chuyển (phút)
5	trip_speed_kph	Double	Vận tốc trung bình (km/h)
6	day_of_week	String	Thứ trong tuần
7	month_label	String	Tháng (dạng MM/YYYY)

Bảng `analyze_by_time` lưu dữ liệu phân tích theo thời gian (tháng), có các trường dữ liệu như bảng 4.3 sau:

Bảng 4.3: Các trường dữ liệu của bảng `analyze_by_time`

STT	Tên trường	Kiểu dữ liệu	Ý nghĩa
1	year	Int	Năm
2	month	Int	Tháng
3	total_revenue	Double	Doanh thu (\$)
4	avg_distance_km	Double	Quãng đường trung bình (km)
5	avg_duration_minutes	Double	Thời gian di chuyển trung bình (phút)
6	avg_speed_kph	Double	Vận tốc trung bình (km/h)

Bảng `analyze_by_routes` bao gồm các trường thông tin như bảng 4.4, lưu trữ dữ liệu phân tích của từng chặng đường theo từng tháng, phục vụ truy vấn những chặng đường phổ biến nhất cho từng địa điểm.

Bảng 4.4: Các trường dữ liệu của bảng `analyze_by_routes`

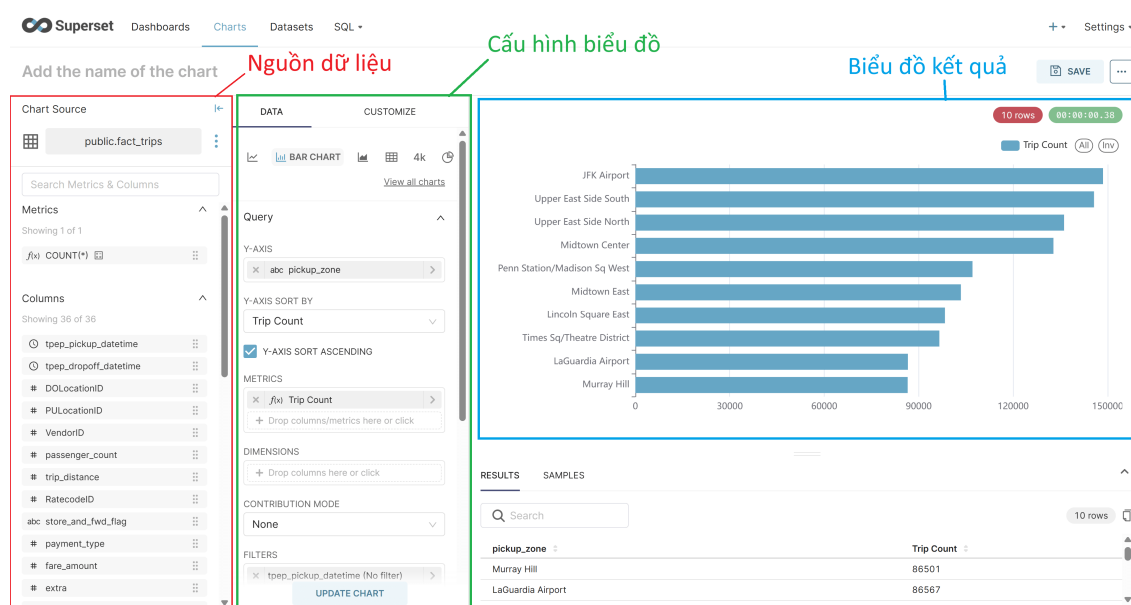
STT	Tên trường	Kiểu dữ liệu	Ý nghĩa
1	year	Int	Năm
2	month	Int	Tháng
3	pickup_zone	String	Điểm đón khách
4	dropoff_zone	String	Điểm trả khách
5	trip_count	Int	Số chuyến đi
6	avg_distance_km	Double	Quãng đường trung bình (km)
7	avg_duration_minutes	Double	Thời gian di chuyển trung bình (phút)
8	avg_fare	Double	Cước phí trung bình (\$)
9	total_revenue	Double	Tổng doanh thu (\$)

4.1.6 Mô-đun trực quan hóa dữ liệu

Sau khi dữ liệu được xử lý và lưu trữ vào các hệ thống cơ sở dữ liệu, bước tiếp theo là trực quan hóa để biến dữ liệu thô thành thông tin hữu ích, phục vụ cho nhu cầu phân tích nghiệp vụ. Mô-đun trực quan hóa trong hệ thống được em triển khai dưới hai hình thức chính: (i) Tạo báo cáo phân tích dữ liệu theo tháng bằng công cụ Superset; (ii) Hiển thị dữ liệu thời gian thực trên giao diện Web, giúp người dùng dễ dàng cập nhật và theo dõi các thông tin cần thiết.

Về Superset, đây là một nền tảng mã nguồn mở mạnh mẽ dành cho việc xây

dựng dashboard và biểu đồ báo cáo, hỗ trợ tích hợp trực tiếp với PostgreSQL thông qua SQLAlchemy URI. Sau khi kết nối cơ sở dữ liệu và tạo một dataset từ bảng trong PostgreSQL, người dùng có thể dễ dàng thiết kế các biểu đồ trực quan tùy chỉnh. Hình 4.9 minh họa ví dụ tạo biểu đồ từ bảng `fact_trips` trong Superset. Giao diện tạo biểu đồ của Superset được chia làm ba vùng chính: (i) Nguồn dữ liệu (màu đỏ): chọn bảng dữ liệu từ hệ thống cơ sở dữ liệu đã kết nối (ví dụ: bảng `public.fact_trips` trong PostgreSQL). Các cột và phép đo (metrics) được liệt kê tại đây. (ii) Cấu hình biểu đồ (màu xanh lá): người dùng chọn loại biểu đồ (bar chart, line chart...), cấu hình trục, nhóm dữ liệu, phép đo, và bộ lọc nếu cần. Dữ liệu được truy vấn SQL tự động từ nguồn tương ứng. (iii) Kết quả biểu đồ (màu xanh dương): hiển thị biểu đồ trực quan theo cấu hình đã chọn. Người dùng có thể quan sát dữ liệu, lưu biểu đồ hoặc thêm vào dashboard.



Hình 4.9: Tạo biểu đồ trong Superset

Bên cạnh khả năng xây dựng báo cáo phân tích, hệ thống còn hỗ trợ trực quan hóa dữ liệu thời gian thực, dữ liệu lịch sử thông qua dịch vụ Web. Em đã triển khai một Web Server đơn giản, với backend sử dụng Spring Boot và frontend được xây dựng bằng ReactJS. Backend có nhiệm vụ kết nối tới Redis và PostgreSQL để truy xuất dữ liệu, sau đó truyền tải dữ liệu về frontend để hiển thị dưới dạng biểu đồ. Giao diện Web được tham khảo từ các nền tảng dịch vụ taxi, tập trung vào việc hiển thị số liệu và biểu đồ minh họa. Mục tiêu của việc tích hợp trực quan hóa trên Web là nhằm thể hiện khả năng mở rộng của hệ thống, có thể dễ dàng phục vụ cho nhiều dịch vụ khác nhau, đặc biệt là các hệ thống cung cấp thông tin theo thời gian thực, gần thời gian thực.

4.2 Triển khai hệ thống

4.2.1 Quy trình và cách thức triển khai ứng dụng trên Kubernetes

Sau khi đã cài đặt môi trường Kubernetes đầy đủ các thành phần từ Control Plane, Worker Node, Load Balancer, Rancher và NFS Server (chi tiết trình bày ở chương 5), bước tiếp theo là triển khai các mô-đun, ứng dụng lên K8s. Có hai cách chính để triển khai ứng dụng lên môi trường K8s, đó là triển khai thủ công và triển khai qua các nền tảng khác như Helm Chart.

Quy trình triển khai các dịch vụ bao gồm các bước chính: (i) Xác định cấu trúc ứng dụng, phân loại thành Stateless (Ví dụ như FastAPI, Spring Boot...) và Stateful (Thường là các cơ sở dữ liệu như Redis, PostgreSQL...); (ii) Đóng gói ứng dụng thành Docker Image và đẩy lên một kho lưu trữ như dockerhub; (iii) Triển khai tài nguyên bằng các file yaml; (iv) Cấu hình Networking, expose dịch vụ qua Ingress hoặc Service; (v) Quản lý storage, gắn dữ liệu với PV, PVC, các biến môi trường qua ConfigMaps; (vi) Giám sát, quản lý tài nguyên cho các ứng dụng.

Một số ứng dụng phổ biến như PostgreSQL, Redis, Superset, Airflow... đã được các nhà cung cấp như Bitnami, Apache, hoặc cộng đồng phát hành dưới dạng Helm Chart. Việc triển khai thông qua Helm giúp giảm thiểu thao tác thủ công, dễ dàng cấu hình theo từng môi trường (values.yaml), đồng thời hỗ trợ rollback và nâng cấp phiên bản một cách thuận tiện. Khi triển khai các ứng dụng theo cách này, ta cần phải nghiên cứu tài liệu để có thể điều chỉnh các thông số trong file values.yaml cho phù hợp, đặc biệt là phần lưu dữ liệu ra bên ngoài bằng PV, PVC.

Để các ứng dụng trong cụm Kubernetes có thể giao tiếp với nhau một cách hiệu quả và thống nhất, Kubernetes cung cấp một cơ chế định danh dịch vụ nội bộ thông qua DNS. Mỗi Deployment, StatefulSet hoặc Service được tạo ra đều có thể được truy cập bởi các ứng dụng khác trong cluster thông qua một địa chỉ theo định dạng chuẩn `<service-name>.<namespace>.svc.cluster.local`, trong đó `service-name` là tên của Service hoặc ứng dụng được định danh, `namespace` là không gian tên (namespace) chứa ứng dụng đó. Ví dụ sau là biến môi trường của Airflow khi triển khai bằng Helm.

```
1 -- values.yaml --
2 env:
3   - name: DATA_INGESTION__TAXI_TYPE
4     value: "yellow"
5   - name: DATA_INGESTION__API_HOST
6     value: "http://data-ingestion-api-server.bigdata.svc.
      cluster.local:5000"
```

```

7  - name: DATA_INGESTION__SPEED
8    value: "1.5"
9  - name: DATA_INGESTION__QUERY_PAGE_SIZE
10   value: "20"
11 - name: KAFKA__BOOTSTRAP_SERVERS
12   value: "kafka.bigdata.svc.cluster.local:9092"
13 - name: SPARK_STREAMING__TRIGGER_TIME
14   value: "3 minutes"
15 - name: HDFS__URI
16   value: "hdfs://hadoop-hadoop-hdfs-nn:9000"
17 - name: REDIS__HOST
18   value: "redis-master.bigdata.svc.cluster.local"
19 - name: REDIS__PORT
20   value: "6379"
21 - name: REDIS__PASSWORD
22   value: "quanda"
23 - name: POSTGRES__URI
24   value: "jdbc:postgresql://postgresql-db.bigdata.svc.
      cluster.local:5432"
25 - name: POSTGRES__USERNAME
26   value: "quanda"
27 - name: POSTGRES__PASSWORD
28   value: "quanda"
29 - name: POSTGRES__DATABASE
30   value: "taxi_trip_db"

```

Trong file `values.yaml`, trường `env` dùng để định nghĩa các biến môi trường của Airflow. Các biến này được khai báo dưới dạng (`name`, `value`), với `name` là tên của biến môi trường, `value` là giá trị của biến đó. Ngoài một số biến môi trường dùng để định nghĩa các giá trị như `DATA_INGESTION__SPEED`, `SPARK_STREAMING__TRIGGER_TIME`, em còn sử dụng biến môi trường để các thành phần có thể giao tiếp với nhau thông qua định danh DNS nội bộ dạng "`<service-name>.<namespace>.svc.cluster.local`". Cách này giúp đảm bảo kết nối ổn định giữa các thành phần trong cùng một namespace (ở đây là `bigdata`) và không bị ảnh hưởng khi địa chỉ IP thay đổi.

4.2.2 Kết quả triển khai

Sau khi triển khai các mô-đun bằng cả hai cách: triển khai thủ công và triển khai bằng Helm, em đã hoàn thiện hệ thống gồm đầy đủ các thành phần như Deployment, StatefulSet, Service, Persistent Volume, Persistent Volume Claim... Hình

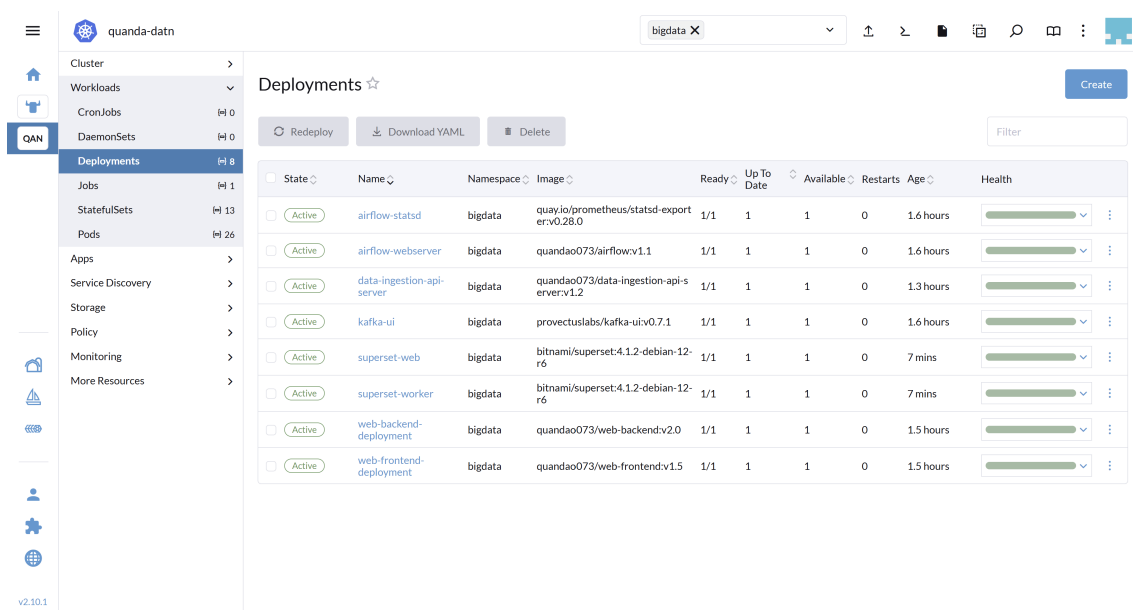
4.10 là danh sách các Pod sau khi triển khai.

```
anhquan@k8s-master-1: / $ kubectl get pod -n bigdata
```

NAME	READY	STATUS	RESTARTS	AGE
airflow-postgresql-0	1/1	Running	0	56m
airflow-scheduler-0	2/2	Running	0	56m
airflow-statsd-78b94c6899-wm4bq	1/1	Running	0	56m
airflow-triggerer-0	2/2	Running	0	56m
airflow-webserver-55bd8455c6-45crl	1/1	Running	0	56m
data-ingestion-api-server-b4d7f4954-mfznf	1/1	Running	0	2m
hadoop-hadoop-hdfs-dn-0	1/1	Running	0	47m
hadoop-hadoop-hdfs-dn-1	1/1	Running	0	47m
hadoop-hadoop-hdfs-dn-2	1/1	Running	0	47m
hadoop-hadoop-hdfs-nn-0	1/1	Running	0	47m
hadoop-hadoop-yarn-nm-0	1/1	Running	1 (46m ago)	47m
hadoop-hadoop-yarn-rm-0	1/1	Running	0	47m
kafka-0	1/1	Running	1 (2m9s ago)	2m33s
kafka-1	1/1	Running	1 (2m8s ago)	2m31s
kafka-2	1/1	Running	1 (2m7s ago)	2m30s
kafka-ui-7cb5f56789-qz7km	1/1	Running	0	2m15s
postgresql-db-0	2/2	Running	0	34m
redis-master-0	1/1	Running	0	39m
spark-master-0	1/1	Running	0	41m
spark-worker-0	1/1	Running	0	41m
spark-worker-1	1/1	Running	0	40m
superset-init-4xnbr	0/1	Completed	3	23m
superset-postgresql-0	1/1	Running	0	23m
superset-redis-master-0	1/1	Running	0	23m
superset-web-6cf58bc675-m7s8l	1/1	Running	0	23m
superset-worker-58fcc696db-gwtfl	1/1	Running	2 (18m ago)	23m
web-backend-deployment-84bd5b4d74-xqjfl	1/1	Running	0	13m
web-frontend-deployment-5bf646d49d-7m9p9	1/1	Running	0	13m
zookeeper-0	1/1	Running	0	3m35s

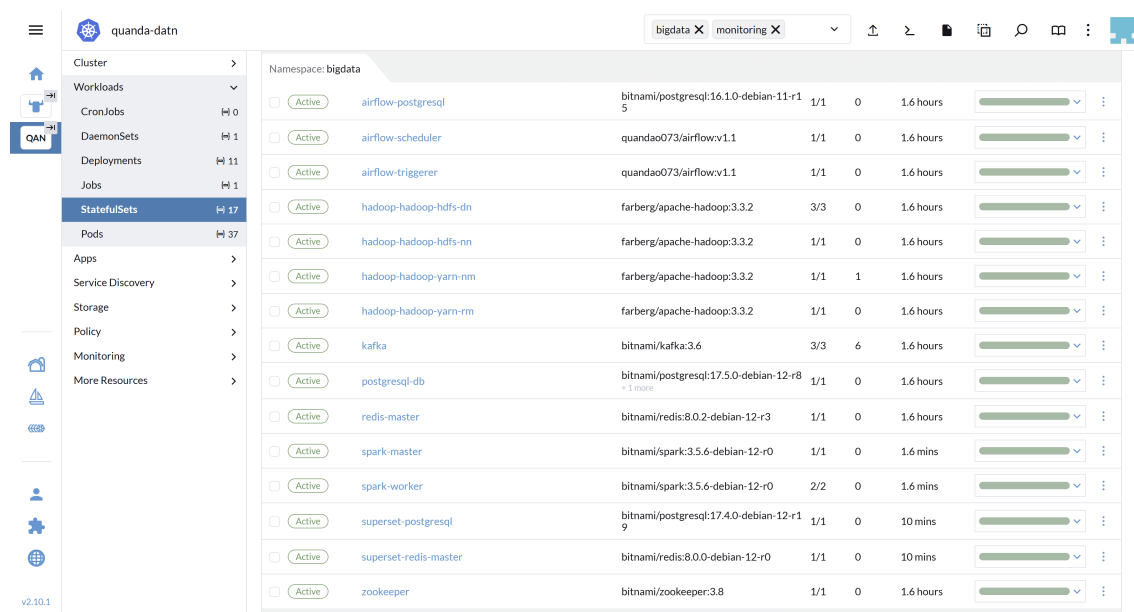
Hình 4.10: Các Pod của hệ thống

Hình 4.11 là kết quả sau khi triển khai các Deployment để quản lý và tự động duy trì số lượng Pod cần thiết cho các thành phần không trạng thái của hệ thống.



Hình 4.11: Các Deployment của hệ thống

Hình 4.12 là danh sách các StatefulSet. Sử dụng StatefulSet để triển khai các dịch vụ có trạng thái, như cơ sở dữ liệu, với định danh và thứ tự khởi động ổn định.

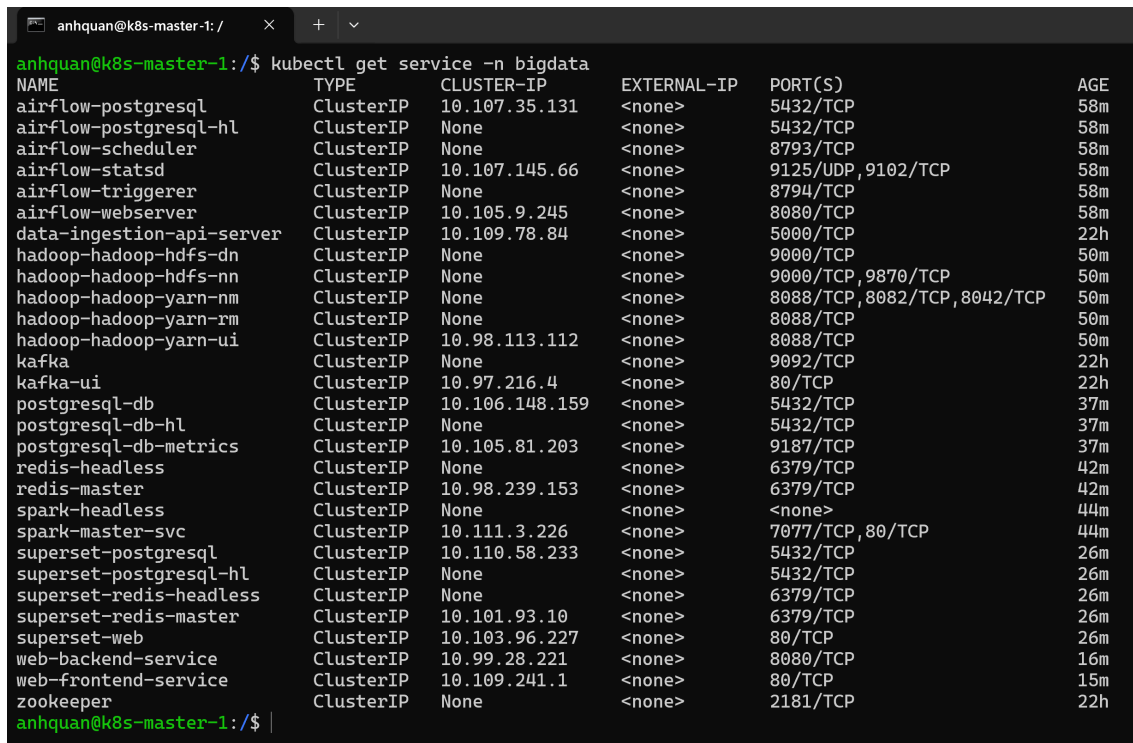


Name	Namespace	Replicas	Ready	Age	Status
airflow-postgresql	bigdata	1/1	0	1.6 hours	Active
airflow-scheduler	quandao073/airflowv1.1	1/1	0	1.6 hours	Active
airflow-triggerer	quandao073/airflowv1.1	1/1	0	1.6 hours	Active
hadoop-hadoop-hdfs-dn	farberg/apache-hadoop:3.3.2	3/3	0	1.6 hours	Active
hadoop-hadoop-hdfs-nn	farberg/apache-hadoop:3.3.2	1/1	0	1.6 hours	Active
hadoop-hadoop-yarn-nm	farberg/apache-hadoop:3.3.2	1/1	1	1.6 hours	Active
hadoop-hadoop-yarn-rm	farberg/apache-hadoop:3.3.2	1/1	0	1.6 hours	Active
kafka	bitnami/kafka:3.6	3/3	6	1.6 hours	Active
postgresql-db	bitnami/postgresql:17.5.0-debian-12-r8	1/1	0	1.6 hours	Active
redis-master	bitnami/redis:8.0.2-debian-12-r3	1/1	0	1.6 hours	Active
spark-master	bitnami/spark:3.5.6-debian-12-r0	1/1	0	1.6 mins	Active
spark-worker	bitnami/spark:3.5.6-debian-12-r0	2/2	0	1.6 mins	Active
superset-postgresql	bitnami/postgresql:17.4.0-debian-12-r19	1/1	0	10 mins	Active
superset-redis-master	bitnami/redis:8.0.0-debian-12-r0	1/1	0	10 mins	Active
zookeeper	bitnami/zookeeper:3.8	1/1	0	1.6 hours	Active

Hình 4.12: Các StatefulSet của hệ thống

Các ứng dụng cần đưa dịch vụ ra bên ngoài như Airflow UI, Kafka UI, Web Frontend sẽ sử dụng Ingress để định tuyến lưu lượng HTTP(S) từ bên ngoài vào đúng dịch vụ bên trong cụm dựa trên tên miền và đường dẫn. Chi tiết sẽ được trình bày ở Chương 5.

Hình 4.13 là danh sách các Service của hệ thống. Service dùng để định nghĩa và cung cấp các điểm truy cập ổn định cho các Pod, hỗ trợ giao tiếp nội bộ giữa các thành phần trong cụm.



```
anhquan@k8s-master-1:/$ kubectl get service -n bigdata
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
airflow-postgresql	ClusterIP	10.107.35.131	<none>	5432/TCP	58m
airflow-postgresql-hl	ClusterIP	None	<none>	5432/TCP	58m
airflow-scheduler	ClusterIP	None	<none>	8793/TCP	58m
airflow-statsd	ClusterIP	10.107.145.66	<none>	9125/UDP, 9102/TCP	58m
airflow-triggerer	ClusterIP	None	<none>	8794/TCP	58m
airflow-webserver	ClusterIP	10.105.9.245	<none>	8080/TCP	58m
data-ingestion-api-server	ClusterIP	10.109.78.84	<none>	5000/TCP	22h
hadoop-hadoop-hdfs-dn	ClusterIP	None	<none>	9000/TCP	50m
hadoop-hadoop-hdfs-nn	ClusterIP	None	<none>	9000/TCP, 9870/TCP	50m
hadoop-hadoop-yarn-nm	ClusterIP	None	<none>	8088/TCP, 8082/TCP, 8042/TCP	50m
hadoop-hadoop-yarn-rm	ClusterIP	None	<none>	8088/TCP	50m
hadoop-hadoop-yarn-ui	ClusterIP	10.98.113.112	<none>	8088/TCP	50m
kafka	ClusterIP	None	<none>	9092/TCP	22h
kafka-ui	ClusterIP	10.97.216.4	<none>	80/TCP	22h
postgresql-db	ClusterIP	10.106.148.159	<none>	5432/TCP	37m
postgresql-db-hl	ClusterIP	None	<none>	5432/TCP	37m
postgresql-db-metrics	ClusterIP	10.105.81.203	<none>	9187/TCP	37m
redis-headless	ClusterIP	None	<none>	6379/TCP	42m
redis-master	ClusterIP	10.98.239.153	<none>	6379/TCP	42m
spark-headless	ClusterIP	None	<none>	<none>	44m
spark-master-svc	ClusterIP	10.111.3.226	<none>	7077/TCP, 80/TCP	44m
superset-postgresql	ClusterIP	10.110.58.233	<none>	5432/TCP	26m
superset-postgresql-hl	ClusterIP	None	<none>	5432/TCP	26m
superset-redis-headless	ClusterIP	None	<none>	6379/TCP	26m
superset-redis-master	ClusterIP	10.101.93.10	<none>	6379/TCP	26m
superset-web	ClusterIP	10.103.96.227	<none>	80/TCP	26m
web-backend-service	ClusterIP	10.99.28.221	<none>	8080/TCP	16m
web-frontend-service	ClusterIP	10.109.241.1	<none>	80/TCP	15m
zookeeper	ClusterIP	None	<none>	2181/TCP	22h

Hình 4.13: Các Service của hệ thống

4.3 Kết quả thực nghiệm

4.3.1 Kết quả thu thập dữ liệu

Hình 4.14 và 4.15 minh họa kết quả của quá trình thu thập dữ liệu. Dữ liệu được đọc từ các file Parquet và giả lập gửi qua API, sau đó được đẩy vào Kafka. Trong Kafka, dữ liệu được phân phối vào 3 phân vùng (partitions) trên 3 broker khác nhau, với hệ số nhân bản (replication factor) là 3 và thời gian lưu giữ (retention time) là 1 ngày.

```
{
  "status": "ok",
  "data": [
    {
      "VendorID": 2,
      "tpep_pickup_datetime": "2024-02-01T00:00:00",
      "tpep_dropoff_datetime": "2024-02-01T22:55:05",
      "passenger_count": 1,
      "trip_distance": 5.25,
      "RatecodeID": 1,
      "store_and_fwd_flag": "N",
      "PULocationID": 68,
      "DOLocationID": 264,
      "payment_type": 2,
      "fare_amount": 28.9,
      "extra": 3.5,
      "mta_tax": 0.5,
      "tip_amount": 0,
      "tolls_amount": 0,
      "improvement_surcharge": 1,
      "total_amount": 33.9,
      "congestion_surcharge": 0,
      "airport_fee": 0
    },
    {
      "VendorID": 2,
      "tpep_pickup_datetime": "2024-02-01T00:00:02",
      "tpep_dropoff_datetime": "2024-02-01T00:25:07",
      "passenger_count": 1,
```

Hình 4.14: Dữ liệu giả lập streaming qua API

Topics

/ yellow_trip_data

Overview

Messages

Consumers

Settings

Statistics

Partitions 3	Replication Factor 3	URP 0	In Sync Replicas 9 of 9	Type External	Segment Size 12 MB	Segment Count 9
Message Count 7393						

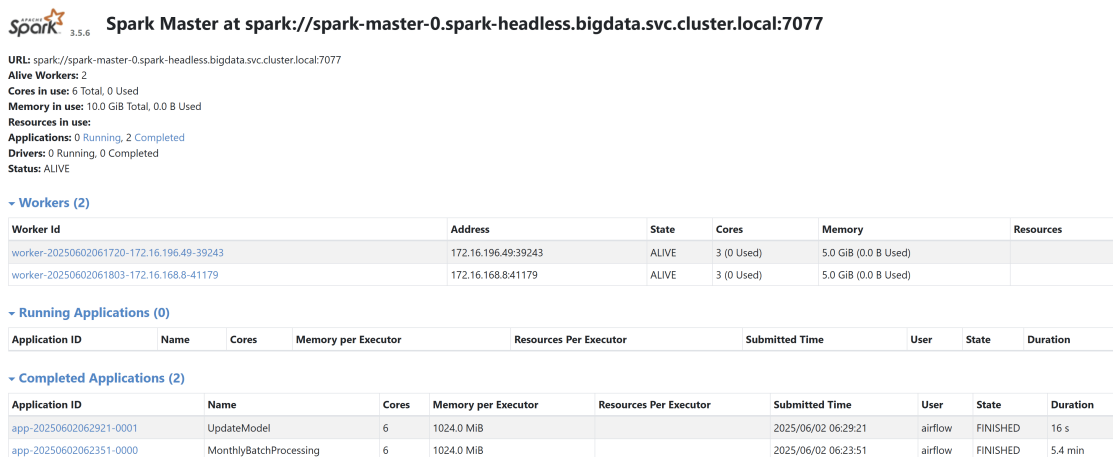
Partition ID	Replicas	First Offset	Next Offset	Message Count
0	1, 2, 0	1589	3847	2258
1	0, 1, 2	1422	3516	2094
2	2, 0, 1	2049	5090	3041

Hình 4.15: Gửi dữ liệu qua Kafka

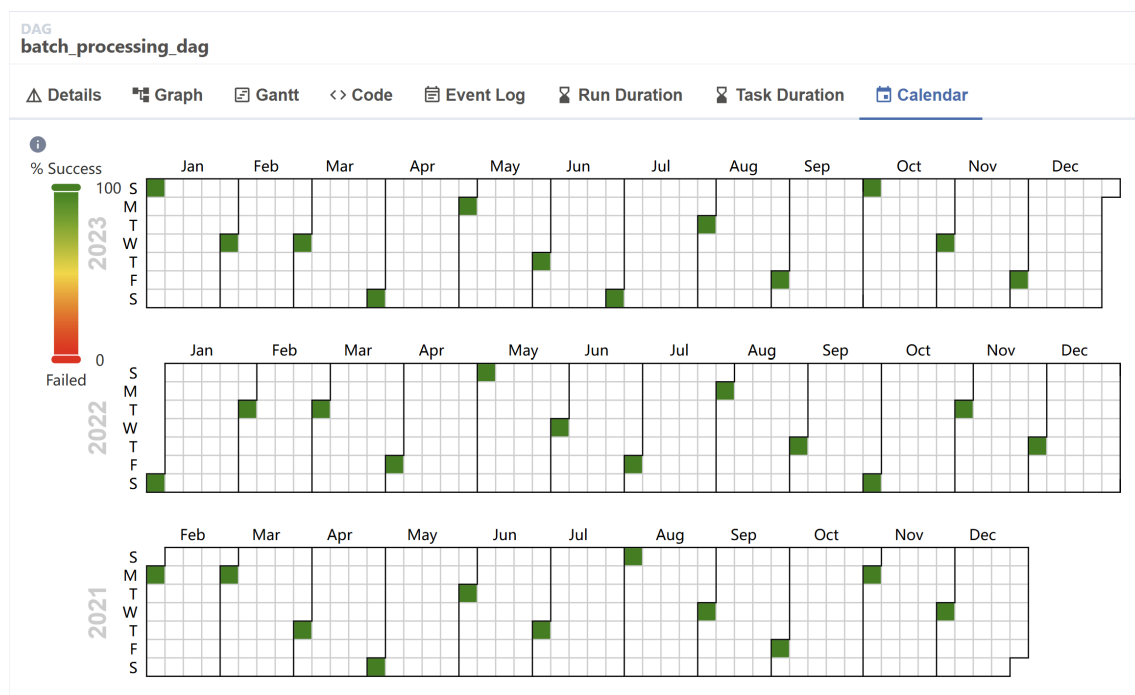
4.3.2 Kết quả xử lý dữ liệu

Hình 4.8 thể hiện kết quả xử lý dữ liệu thời gian thực bằng Spark Structured Streaming, sau đó gửi đến kênh Redis Pub/Sub để phục vụ trực quan hóa tức thời trên giao diện Web.

Đối với xử lý dữ liệu theo lô, hình 4.16 minh họa kết quả của từng chu kỳ batch. Mỗi chu kỳ bao gồm hai giai đoạn chính: (i) Biến đổi và chuẩn hóa dữ liệu, (ii) Cập nhật và huấn luyện mô hình dự đoán dựa trên dữ liệu lịch sử. Việc thực hiện xử lý dữ liệu được tự động bởi Airflow, xử lý theo lô mỗi tháng một lần, kết quả như hình 4.17.



Hình 4.16: Xử lý dữ liệu theo lô bằng Spark SQL

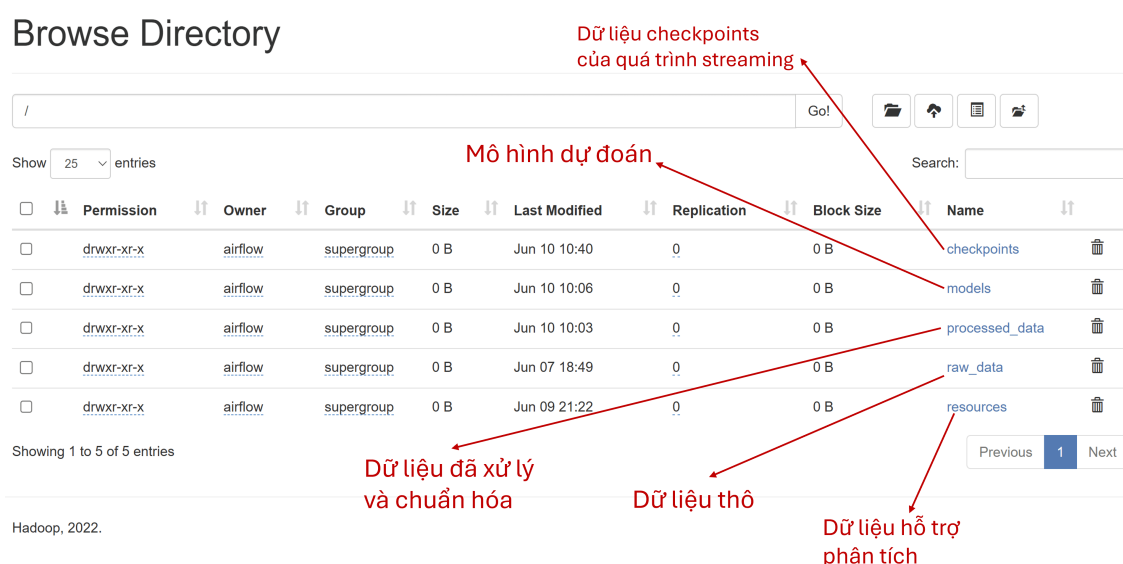


Hình 4.17: Lập lịch tác vụ trên Airflow

4.3.3 Kết quả lưu trữ dữ liệu

Dữ liệu thời gian thực được ghi vào kênh Redis Pub/Sub dưới định dạng JSON, bao gồm các trường: thời gian hiện tại, số lượng chuyển đi thực tế, thời điểm dự đoán và số lượng chuyển đi dự đoán trong một giờ tiếp theo (xem hình 4.8).

Hình 4.18 minh họa hệ thống lưu trữ trên HDFS, bao gồm: dữ liệu thô, dữ liệu đã được xử lý và chuẩn hóa, mô hình dự đoán đã huấn luyện, dữ liệu hỗ trợ phân tích, cùng với checkpoint của quá trình streaming.



Hình 4.18: Cấu trúc thư mục trên HDFS

Với PostgreSQL, hình 4.19 thể hiện cấu trúc bảng cơ sở dữ liệu, bao gồm: (i) Dữ liệu đã được xử lý để tạo báo cáo phân tích trong Superset, (ii) Dữ liệu phục vụ các truy vấn tổng hợp và thời gian thực trên giao diện Web.

```

postgresdb-0 [X] [^]
Type "help" for help.

taxi_trip_db=> \dt
                List of relations
 Schema |      Name      | Type  | Owner
-----+-----+-----+-----
 public | analyze_by_routes | table | quanda
 public | analyze_by_time   | table | quanda
 public | fact_trips        | table | quanda
(3 rows)

taxi_trip_db=> \d analyze_by_routes
                Table "public.analyze_by_routes"
   Column      |      Type      | Collation | Nullable | Default
-----+-----+-----+-----+-----
 year           | integer         |           |          |
 month          | integer         |           |          |
 pickup_zone    | text            |           |          |
 dropoff_zone   | text            |           |          |
 trip_count     | bigint          |           | not null |
 avg_distance_km | double precision |           |          |
 avg_duration_minutes | double precision |           |          |
 avg_fare       | double precision |           |          |
 total_revenue  | double precision |           |          |

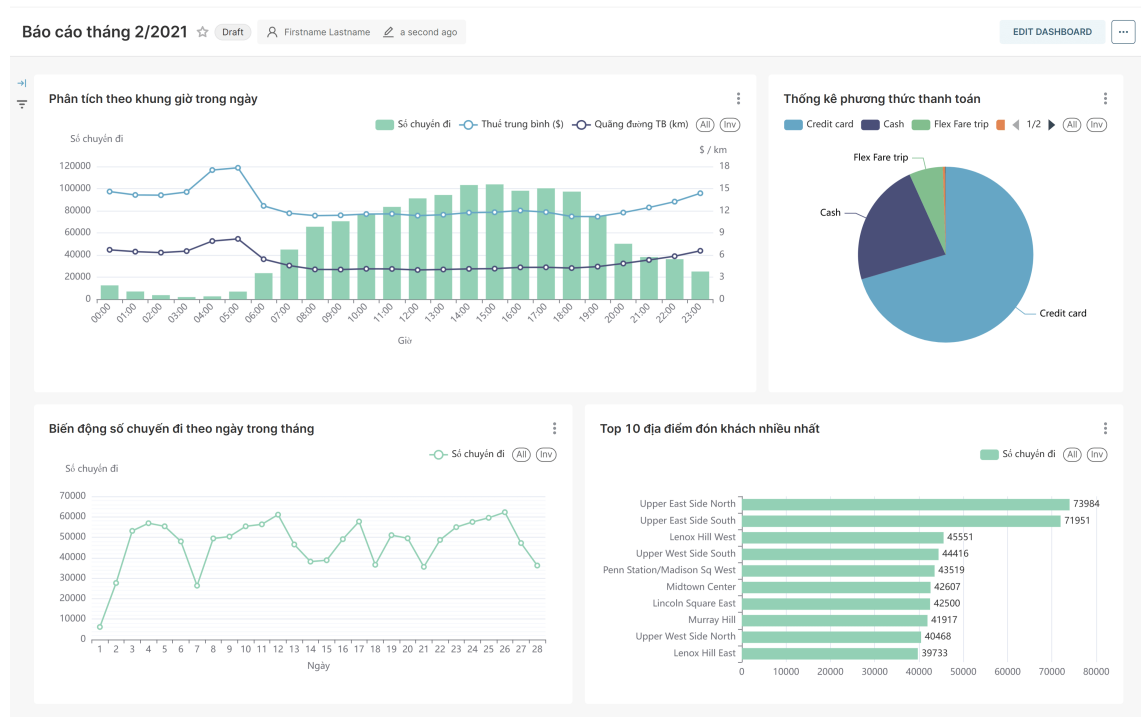
taxi_trip_db=> \d analyze_by_time
                Table "public.analyze_by_time"
   Column      |      Type      | Collation | Nullable | Default
-----+-----+-----+-----+-----
 year           | integer         |           |          |
 month          | integer         |           |          |
 trip_count     | bigint          |           | not null |
 total_revenue  | double precision |           |          |
 avg_distance_km | double precision |           |          |
 avg_duration_minutes | double precision |           |          |
 avg_speed_kph  | double precision |           |          |

```

Hình 4.19: Cấu trúc bảng trong PostgreSQL

4.3.4 Kết quả trực quan hóa dữ liệu

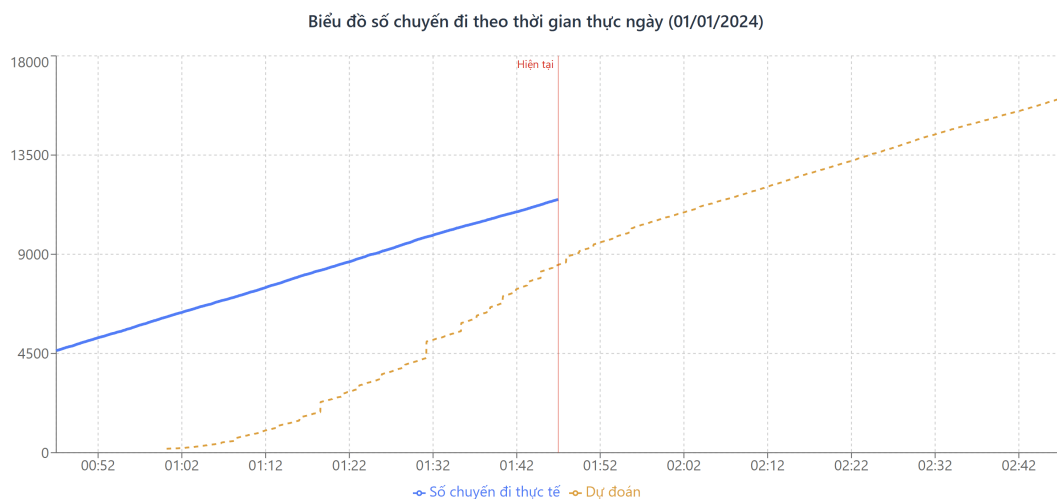
Hình 4.20 minh họa một báo cáo tùy chỉnh được tạo bằng Superset với dữ liệu của một tháng. Người dùng có thể linh hoạt thiết kế các biểu đồ phân tích theo thời gian, khu vực, loại hình thanh toán hoặc doanh thu, từ đó hỗ trợ các quyết định chiến lược về quản lý và kinh doanh.



Hình 4.20: Tạo báo cáo bằng Superset

Bên cạnh Superset, hệ thống còn tích hợp một ứng dụng Web trực quan hóa dữ liệu thời gian thực. Giao diện Web cho phép người dùng theo dõi các chỉ số như lượng chuyến đi, dự báo trong tương lai, phân tích theo tuyến đường và mốc thời gian. Kết quả được thể hiện thông qua các hình 4.21, 4.22 và 4.23.

Dữ liệu thời gian thực



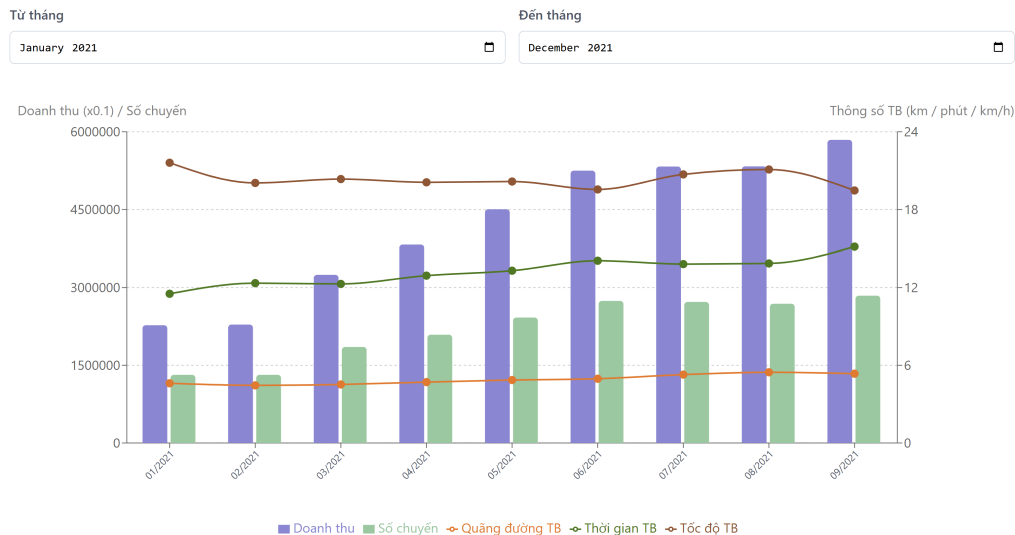
Hình 4.21: Biểu đồ số chuyến đi theo thời gian thực

Thống kê tuyến đường phổ biến theo địa điểm

Chọn tháng		Chọn địa điểm			
January 2021		Upper East Side North			
Điểm đón khách	Điểm trả khách	Số chuyến	Doanh thu (\$)	Khoảng cách TB (km)	Thời gian TB (phút)
Upper East Side North	Upper East Side South	9905	114897.64	1.71	6.69
Upper East Side North	East Harlem South	4132	43933.24	1.55	5.69
Upper East Side North	Lenox Hill West	3956	48363.80	1.87	7.79
Upper East Side North	Upper West Side North	3222	41398.81	2.33	7.83
Upper East Side South	Upper East Side North	11750	133703.72	1.72	6.12
Lenox Hill West	Upper East Side North	4834	55027.01	1.81	6.37
Yorkville West	Upper East Side North	4001	39824.47	1.11	4.75
Lenox Hill East	Upper East Side North	3694	54215.96	2.11	8.29

Hình 4.22: Thống kê tuyến đường phổ biến theo địa điểm

Thống kê dữ liệu theo tháng



Hình 4.23: Thống kê dữ liệu theo tháng

4.3.5 Khả năng chịu lỗi

Hình 4.24 cho thấy khả năng chịu lỗi của cụm Kafka. Phần trên của hình cho thấy cụm Kafka đang hoạt động ổn định với 3 broker (ID 0, 1, 2), tất cả đều online và mỗi broker có 3 phân vùng (partition) online, 1 partition leader, và hệ thống có 9 bản sao đang đồng bộ (In Sync Replicas). Active Controller tại thời điểm này là broker ID 2.

Phần dưới của hình thể hiện trạng thái sau khi broker ID 2 bị xóa (tạm thời ngắt kết nối). Cụm Kafka vẫn duy trì được trạng thái hoạt động: 3 partition vẫn online,

nhưng một số bản sao rơi vào trạng thái không đồng bộ (Out Of Sync Replicas = 3), và có 3 bản sao chưa đồng bộ hoàn toàn (URP = 3).

Đặc biệt, cột Active Controller chuyển thành 0, thể hiện rằng hệ thống đã tự động bầu lại một broker khác (ở đây là broker ID 0) làm controller mới thay cho broker vừa mất kết nối. Điều này minh chứng cho khả năng tự động phục hồi (self-healing) và duy trì tính sẵn sàng (availability) của Kafka dù xảy ra sự cố một nút trong cụm.

Khi broker ID 2 được khởi động lại, hệ thống sẽ tự động đồng bộ lại các bản sao, cập nhật lại danh sách In-Sync Replicas, và tiếp tục hoạt động bình thường mà không làm gián đoạn dòng dữ liệu.

Brokers							
Uptime			Partitions				
Broker Count	Active Controller	Version	Online	URP	In Sync Replicas	Out Of Sync Replicas	
3	2	3.6-IV2	3 of 3	0	9 of 9	0	
Broker ID	Disk usage	Partitions skew	Leaders	Leader skew	Online partitions	Port	Host
0	2 MB, 3 segment(s)	-	1	-	3	9092	kafka-0.kafka.bigdata.svc.cluster.local
1	2 MB, 3 segment(s)	-	1	-	3	9092	kafka-1.kafka.bigdata.svc.cluster.local
2	2 MB, 3 segment(s)	-	1	-	3	9092	kafka-2.kafka.bigdata.svc.cluster.local

Brokers							
Uptime			Partitions				
Broker Count	Active Controller	Version	Online	URP	In Sync Replicas	Out Of Sync Replicas	
2	0	3.6-IV2	3 of 3	3	6 of 9	3	
Broker ID	Disk usage	Partitions skew	Leaders	Leader skew	Online partitions	Port	Host
0	2 MB, 3 segment(s)	-	2	-	3	9092	kafka-0.kafka.bigdata.svc.cluster.local
1	2 MB, 3 segment(s)	-	1	-	3	9092	kafka-1.kafka.bigdata.svc.cluster.local

Hình 4.24: Khả năng chịu lỗi của cụm Kafka

Khả năng chịu lỗi của hệ thống còn được thể hiện qua việc đưa dữ liệu của các dịch vụ ra bên ngoài bằng NFS-Server, đảm bảo rằng khi dịch vụ có gặp sự cố thì dữ liệu vẫn luôn tồn tại trên NFS-Server. Đồng thời, em cũng triển khai thêm tính năng sao lưu dữ liệu của cụm Kubernetes bằng Velero [18], lưu trữ trên MinIO [19]. Chỉ cần một vài thao tác đơn giản là có thể khôi phục lại trạng thái của cụm như ban đầu. Chi tiết phần này sẽ được trình bày ở Chương 5.

4.3.6 Khả năng mở rộng

Hệ thống được xây dựng theo hướng module hóa và triển khai trên nền tảng Kubernetes nên dễ dàng mở rộng theo cả chiều ngang (scale-out) và chiều dọc (scale-up). Các thành phần xử lý như Spark, Kafka, API server đều được đóng gói dưới dạng container và triển khai thành các pod độc lập, có thể dễ dàng nhân bản

để tăng công suất xử lý khi khối lượng dữ liệu tăng lên. Kubernetes hỗ trợ cơ chế autoscaling theo tài nguyên sử dụng hoặc số lượng request, giúp hệ thống tự động thích nghi với tải thực tế.

CHƯƠNG 5. CÁC GIẢI PHÁP VÀ ĐÓNG GÓP NỔI BẬT

5.1 Giới thiệu vấn đề

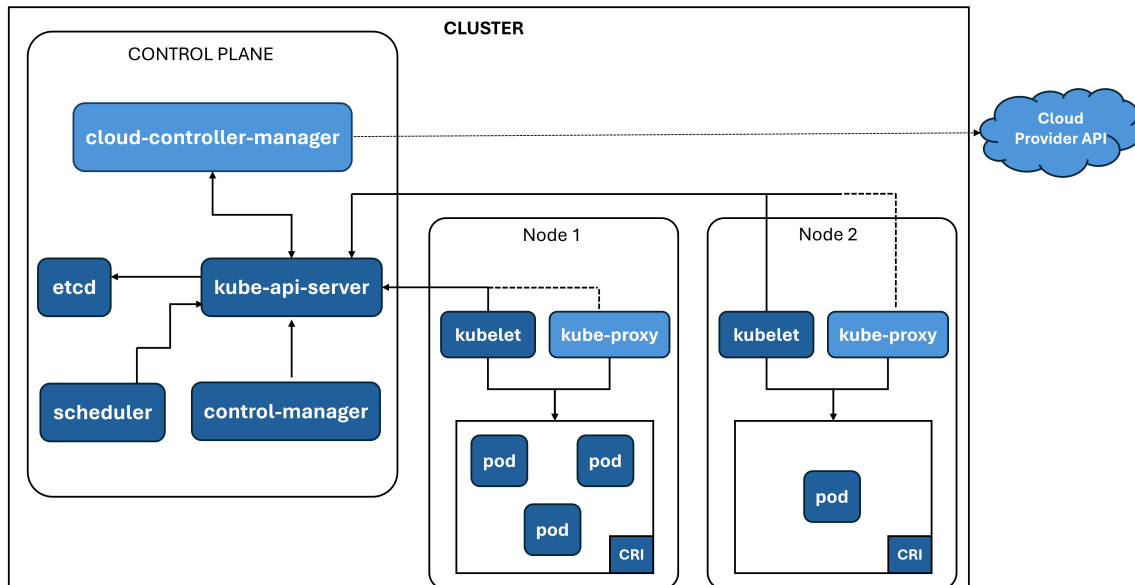
Như đã trình bày ở phần 3.7, Kubernetes (hay K8s) là một nền tảng triển khai các ứng dụng container được các doanh nghiệp ưa chuộng với khả năng tự động hóa, mở rộng linh hoạt, quản lý các dịch vụ, ứng dụng một cách chuyên nghiệp, đảm bảo tính bảo mật, chịu lỗi cao. Việc triển khai một hệ thống xử lý dữ liệu lớn trên nền tảng Kubernetes (K8s) thường được hướng dẫn bằng các công cụ đơn giản như Minikube, Kind hoặc K3s. Tuy nhiên, những công cụ này chỉ phù hợp để thử nghiệm cục bộ, không đáp ứng được các tiêu chí triển khai hệ thống thực tế như: mở rộng, phân quyền, cân bằng tải, phân tách tài nguyên và khả năng giám sát chuyên sâu. Khi triển khai trên môi trường Cloud, một số dịch vụ sẽ được cung cấp sẵn như các lớp lưu trữ (Storage Classes), quản lý Traffic, etcd HA, Network Overlay, công cụ giám sát... Nhưng khi triển khai trên môi trường On-Premise, ta cần phải tự cài đặt, cấu hình mọi thứ. Không chỉ vậy, việc triển khai các ứng dụng như thế nào để đảm bảo các yếu tố kể trên, tiếp cận một cách tổng quan để triển khai tất cả các ứng dụng nói chung và các ứng dụng dữ liệu lớn nói riêng đòi hỏi rất nhiều thời gian tìm hiểu, nghiên cứu, thử nghiệm... Vì vậy, chương này em sẽ trình bày một cách tiếp cận trong việc triển khai một cụm K8s trong môi trường On-Premise sao cho chuyên nghiệp và hiệu quả nhất. Kiến trúc hệ thống này được tham khảo từ Devopsedu.vn [20], trong đó đã phát triển để triển khai các hệ thống phân tán phục vụ cho dữ liệu lớn.

5.2 Thiết kế kiến trúc môi trường triển khai Kubernetes

5.2.1 Cụm máy chủ Kubernetes

Hình 5.1 mô tả kiến trúc tiêu chuẩn của một cụm Kubernetes. Ở đó, một cụm máy chủ Kubernetes bao gồm Control Plane (hay còn gọi là Master Node), và các Worker Node. Control Plane là bộ não của cụm K8s, chịu trách nhiệm vận hành và quản lý trạng thái của cụm, bao gồm các thành phần (i) `kube-apiserver`, tiếp nhận yêu cầu, xử lý logic và điều phối các tác vụ cho Control Plane; (ii) `scheduler`, phân bổ công việc mới cho các Worker Node một cách phù hợp, đảm bảo cân bằng giữa các node; (iii) `controller-manager`, đây là thành phần thực thi các tiến trình điều khiển và điều tiết trạng thái cụm; (iv) `etcd`, kho lưu trữ của cụm, chứa các thông tin về cấu hình và trạng thái cụm dưới dạng key-value phân tán. Các Worker Node thì khác, đóng vai trò chạy các workload (pod) của ứng dụng. Mỗi Worker Node sẽ bao gồm kubelet - quản lý vòng đời của các Pod trong Node, kube-proxy - cung cấp dịch vụ mạng cho Pod và Container Runtime -

chạy các Container trong các Pod.



Hình 5.1: Kiến trúc của một cụm Kubernetes

Từ kiến trúc đó, mô hình thông thường sẽ là 1 Control Plane và nhiều Worker Node. Mô hình truyền thống này có một nhược điểm là không đảm bảo tính sẵn sàng, vì khi Control Plane gặp sự cố thì sẽ khiến cả cụm K8s dừng hoạt động. Trong dự án này, em lựa chọn một kiến trúc mới hơn: triển khai cụm Kubernetes với ba máy chủ Ubuntu, trong đó cả ba đều đảm nhiệm vai trò Control Plane (Master Node) đồng thời cũng là các Node thực thi (Worker Node). Cấu trúc này giúp nâng cao tính sẵn sàng (High Availability – HA), vì hệ thống vẫn có thể duy trì hoạt động nếu một trong ba máy chủ gặp lỗi. Đồng thời, giải pháp này rất phù hợp với môi trường triển khai on-premise quy mô nhỏ, nơi việc tách biệt hoàn toàn giữa Master và Worker dễ gây lãng phí tài nguyên. Cả 3 node được kết nối qua mạng nội bộ tốc độ cao, chia sẻ các tệp cấu hình `kubeconfig`, `certificates` để đảm bảo đồng bộ bảo mật. Cụm master hoạt động với địa chỉ ảo thông qua `keepalived` và proxy qua `nginx`, cho phép truy cập API Server qua một địa chỉ duy nhất ngay cả khi một node bị lỗi. Với mô hình này, ta có thể dễ dàng thêm các Node khác vào cụm, có thể là Worker Node hoặc Master Node, sao cho phù hợp với tài nguyên sẵn có.

5.2.2 Giao diện quản lý - Rancher

Thông thường khi thao tác trên cụm K8s, ta cần chạy các lệnh thực thi sử dụng `kubectl`. Việc này gây ra rất nhiều bất tiện, ta không thể giám sát các Pod, Node một cách dễ dàng được. Khi muốn chỉnh sửa hay cập nhật cũng rất khó khăn. Vì vậy, em đề xuất sử dụng nền tảng mã nguồn mở Rancher [21] để làm giao diện quản lý trực quan cho cụm Kubernetes.

Rancher là một nền tảng quản lý Kubernetes toàn diện, cung cấp giao diện web trực quan giúp người dùng dễ dàng theo dõi, điều phối và cấu hình các tài nguyên trong cụm (cluster). Thông qua Rancher, người quản trị có thể quan sát trạng thái của các Pod, Node, Namespace, Deployment và các thành phần liên quan một cách trực quan; thực hiện phân quyền người dùng dựa trên dự án và namespace bằng cơ chế RBAC; chỉnh sửa hoặc tạo mới các file cấu hình YAML và triển khai ứng dụng trực tiếp trên giao diện; đồng thời theo dõi log cũng như các sự kiện (event) của các Pod một cách tập trung. Ngoài ra, Rancher còn hỗ trợ khả năng quản lý nhiều cụm Kubernetes cùng lúc, giúp việc vận hành các hệ thống phân tán trở nên linh hoạt và hiệu quả hơn.

Status	Name	Namespace	Image	Replicas	Version	IP	Node	Age
Running	spark-master-0	bigdata	bitnami/spark:3.5.5-debian-12-r8	1/1	1 (4h38m ago)	172.16.182.120	k8s-master-2	5 days
Running	spark-worker-0	bigdata	bitnami/spark:3.5.5-debian-12-r8	1/1	1 (4h38m ago)	172.16.182.121	k8s-master-2	5 days
Running	spark-worker-1	bigdata	bitnami/spark:3.5.5-debian-12-r8	1/1	1 (4h38m ago)	172.16.196.14	k8s-master-1	5 days
Completed	superset-init-t7b2t	bigdata	bitnami/superset:4.1.2-debian-12-r6	0/1	4	<none>	k8s-master-3	6 days
Running	superset-postgresql-0	bigdata	bitnami/postgresql:17.4.0-debian-12-r19	1/1	2 (4h38m ago)	172.16.196.7	k8s-master-1	6 days
Running	superset-redis-master-0	bigdata	bitnami/redis:8.0.0-debian-12-r0	1/1	2 (4h38m ago)	172.16.196.4	k8s-master-1	6 days
Running	superset-web-6d6676fdb9-h59t9	bigdata	bitnami/superset:4.1.2-debian-12-r6	1/1	2 (4h38m ago)	172.16.196.23	k8s-master-1	6 days
Running	superset-worker-b665466b-f8tp	bigdata	bitnami/superset:4.1.2-debian-12-r6	1/1	6 (4h38m ago)	172.16.168.39	k8s-master-3	6 days
Running	velero-7974469749-qfkyz	velero	velero/velero:v1.16.1	1/1	2 (4h38m ago)	172.16.168.44	k8s-master-3	6 days
Terminating	web-backend-deployment-588d946cdb-24r2q	bigdata	quanda073/web-backend:v2.0	0/1	0	<none>	k8s-master-2	1.1 mins
Containers with unready status: [web-backend]								
Running	web-backend-deployment-588d946cdb-89j8r	bigdata	quanda073/web-backend:v2.0	1/1	6 (4h38m ago)	172.16.168.53	k8s-master-3	6 days
ContainerCreating	web-backend-deployment-588d946cdb-gl7q	bigdata	quanda073/web-backend:v2.0	0/1	0	<none>	k8s-master-1	1.1 mins
Containers with unready status: [web-backend]								
Running	web-frontend-deployment-698dbb4cb-7dcmh	bigdata	quanda073/web-frontend:v1.5.1	1/1	1 (4h38m ago)	172.16.196.25	k8s-master-1	5 days
Running	zookeeper-0	bigdata	bitnami/zookeeper:3.8	1/1	1 (4h38m ago)	172.16.196.38	k8s-master-1	5 days

Hình 5.2: Giao diện quản lý của Rancher

Hình 5.2 là ví dụ về giao diện quản lý của Rancher. Có thể thấy thông tin về các Pod được hiển thị rất trực quan, bao gồm trạng thái (Running, Terminating...), thông tin về Namespace, Image, Node, số lần chạy lại. Ngoài ra, các tài nguyên khác như Deployment, StatefulSet... cũng được quản lý tương tự.

Trong hệ thống của em, Rancher được cài đặt trên một node riêng và kết nối trực tiếp đến API Server của cụm. Điều này giúp tách biệt lớp quản trị với lớp xử lý, nâng cao bảo mật và đảm bảo hiệu năng hoạt động. Ngoài ra có thể quản lý cụm từ xa bằng cách kết nối đến máy chủ chạy Rancher. Giao diện Rancher hoạt động ổn định, phản hồi nhanh và hỗ trợ hiệu quả việc theo dõi tiến trình xử lý dữ liệu trong từng Pod của hệ thống như Spark Executor, Kafka Broker hay các tác vụ batch từ Airflow.

Việc sử dụng Rancher không chỉ giúp đơn giản hóa quá trình quản trị cụm K8s,

mà còn giảm thời gian triển khai và tăng khả năng tương tác của người dùng với hệ thống mà không cần kiến thức sâu về câu lệnh `kubectl`.

5.2.3 Thành phần cân bằng tải - Nginx

Trong một hệ thống Kubernetes on-premise, khi triển khai các dịch vụ cần truy cập từ bên ngoài (như Superset, Grafana, Airflow...), ta cần một thành phần đóng vai trò như là “người bảo vệ”, cổng giao tiếp giữa người dùng và hệ thống bên trong. Thành phần đó chính là Ingress Controller kết hợp với một máy chủ cân bằng tải. Trong đề án này, em triển khai một node riêng biệt để làm Load Balancer, sử dụng phần mềm mã nguồn mở Nginx để đảm nhiệm vai trò đó.

Về bản chất, NGINX hoạt động như một `reverse proxy` – nghĩa là nó nhận các yêu cầu HTTP/HTTPS từ người dùng bên ngoài, sau đó định tuyến tới đúng Pod hoặc dịch vụ nội bộ trong cụm K8s. Bằng cách này, người dùng chỉ cần truy cập qua một địa chỉ duy nhất (VD: `quanda.superset.local`) và Nginx sẽ chịu trách nhiệm đưa các yêu cầu vào đúng nơi cần xử lý.

Trong kiến trúc hệ thống, NGINX được kết hợp với Ingress Controller bên trong cụm Kubernetes. Cấu hình Ingress được định nghĩa bằng YAML, chỉ rõ đường dẫn, tên host và dịch vụ nội bộ tương ứng. Ví dụ như trong hình 5.3, các thành phần như `hadoop-namenode`, `kafka-ui`, `superset`, `web-frontend` có thể truy cập qua các tên miền như `http://quanda.hadoop.local/...`

State	Name	Target	Default	Ingress Class	Age
Namespace: bigdata					
Active	airflow-ingress	http://quanda.airflow-web.local/ > airflow-webserver	—	nginx	1.6 hours
Active	hadoop-namenode-ingress	http://quanda.hadoop.local/ > hadoop-hadoop-hdfs-nn	—	nginx	23 hours
Active	kafka-ui-ingress	http://quanda.kafka-ui.local/ > kafka-ui	—	nginx	23 hours
Active	spark-ingress	http://quanda.spark-ui.local/ > spark-master-svc	—	nginx	1 mins
Active	superset-web	http://quanda.superset.local/ > superset-web	—	nginx	9 mins
Active	web-backend-ingress	http://api-quanda.web-backend.local/ > web-backend-service	—	nginx	23 hours
Active	web-frontend-ingress	http://quanda.web-frontend.local/ > web-frontend-service	—	nginx	23 hours
Namespace: monitoring					
Active	grafana-ingress	http://quanda.grafana.local/ > quanda-grafana	—	nginx	7 mins
Active	prometheus-ingress	http://quanda.prometheus.local/ > prometheus-operated	—	nginx	7 mins

Hình 5.3: Các dịch vụ được truy cập qua Nginx-Ingress

Việc triển khai Load Balancer riêng biệt tách biệt hẳn lớp truy cập với phần hạ tầng xử lý, giúp tăng tính bảo mật, dễ cấu hình và mở rộng về sau. Đồng thời giúp đơn giản hóa quá trình cấu hình DNS nội bộ – chỉ cần trỏ các subdomain về địa chỉ

IP của node này là có thể truy cập các dịch vụ một cách dễ dàng.

5.2.4 NFS Server - Lưu trữ dữ liệu

Trong hệ thống Kubernetes, việc sử dụng các loại Persistent Volume (PV) là cần thiết để lưu trữ dữ liệu lâu dài cho các ứng dụng như PostgreSQL, Redis, Superset hay các job batch. Trên môi trường Cloud, người dùng có thể dễ dàng sử dụng các dịch vụ lưu trữ như EBS (AWS), PD (GCP) hay CSI driver (Azure). Tuy nhiên, với môi trường On-Premise, em triển khai một node riêng để làm nhiệm vụ lưu trữ, sử dụng hệ thống Network File System (NFS).

NFS là một giao thức chia sẻ tệp tin qua mạng được hỗ trợ rộng rãi và dễ tích hợp vào Kubernetes. Trong hệ thống này, node lưu trữ được cấu hình để export một thư mục chính ra toàn bộ các node trong cụm K8s. Các Pod khi tạo PersistentVolumeClaim sẽ thông qua một StorageClass tùy chỉnh, ánh xạ đến một cấu hình PersistentVolume kiểu NFS.

Có hai phương pháp được triển khai để cung cấp lưu trữ qua NFS: (i) Static provisioning sử dụng NFS-Subdir-External-Provisioner: quản trị viên cần tạo thủ công các thư mục con tương ứng với từng PersistentVolume, sau đó khai báo các PV tương ứng trong file YAML. Cách này phù hợp với hệ thống đơn giản, ít thay đổi và có thể kiểm soát tốt việc phân phối thư mục lưu trữ. (ii) Dynamic provisioning với NFS-Subdir-External-Provisioner: sử dụng một StorageClass với driver `nfs-subdir-external-provisioner`, cho phép tự động tạo thư mục con trong thư mục export NFS mỗi khi người dùng khởi tạo PVC. Đây là giải pháp cực kỳ hiệu quả cho các hệ thống dữ liệu lớn hoặc có nhiều namespace, nhiều ứng dụng phân tán. Mỗi ứng dụng có thể tự tạo PVC mà không cần quản trị viên can thiệp.

<input type="checkbox"/> Bound	data-kafka-0	bigdata	Bound	pvc-bfbf5c1c-7a60-445f-a255-58dea01dd557	5Gi	RWO	nfs-kafka-storage	<unset>	Filesystem	23 hours	⋮
<input type="checkbox"/> Bound	data-kafka-1	bigdata	Bound	pvc-35a9f989-9ff3-4c20-a0b7-aa7af7dc247a	5Gi	RWO	nfs-kafka-storage	<unset>	Filesystem	23 hours	⋮
<input type="checkbox"/> Bound	data-kafka-2	bigdata	Bound	pvc-374977bb-4f7c-477f-90c3-63df4780c5c5	5Gi	RWO	nfs-kafka-storage	<unset>	Filesystem	23 hours	⋮
<input type="checkbox"/> Bound	data-source-pvc	bigdata	Bound	data-source-pv	30Gi	ROX	nfs-storage	<unset>	Filesystem	1 day	⋮
<input type="checkbox"/> Bound	data-superset-postgresql-0	bigdata	Bound	pvc-fb2c0ebc-d4d7-46ed-8100-0aaaf368b0a7	8Gi	RWO	nfs-superset-storage	<unset>	Filesystem	1 day	⋮
<input type="checkbox"/> Bound	data-zookeeper-0	bigdata	Bound	pvc-9b2bfd38-53cf-4b6c-807e-093ffa64bc1e	3Gi	RWO	nfs-kafka-storage	<unset>	Filesystem	23 hours	⋮


```

root@nfs-server: /A/kafka# ls
bigdata-data-kafka-0-pvc-bfbf5c1c-7a60-445f-a255-58dea01dd557
bigdata-data-kafka-1-pvc-35a9f989-9ff3-4c20-a0b7-aa7af7dc247a
bigdata-data-kafka-2-pvc-374977bb-4f7c-477f-90c3-63df4780c5c5
bigdata-data-source-pvc-pvc-fb2c0ebc-d4d7-46ed-8100-0aaaf368b0a7
bigdata-data-superset-postgresql-0-pvc-fb2c0ebc-d4d7-46ed-8100-0aaaf368b0a7
bigdata-data-zookeeper-0-pvc-9b2bfd38-53cf-4b6c-807e-093ffa64bc1e
root@nfs-server: /A/kafka#

```

Hình 5.4: Các Persistent Volume Claim của cụm Kafka

Hình 5.4 là một sơ đồ PVC của các ứng dụng. Ví dụ Kafka có một lớp lưu trữ là `nfs-kafka-storage` kiểu NFS động gắn với thư mục `/A/kafka` trên Node NFS Server, đây là nơi lưu trữ toàn bộ dữ liệu của cụm Kafka. Khi ta tăng số lượng broker của Kafka lên thì hệ thống sẽ tự động tạo ra một thư mục để lưu dữ liệu của broker đó.

Việc sử dụng node NFS riêng biệt giúp phân tách hoàn toàn nhiệm vụ lưu trữ với các chức năng khác của hệ thống, đồng thời đảm bảo hiệu suất ổn định cho các ứng dụng xử lý dữ liệu lớn chạy song song.

5.3 Kết quả đạt được

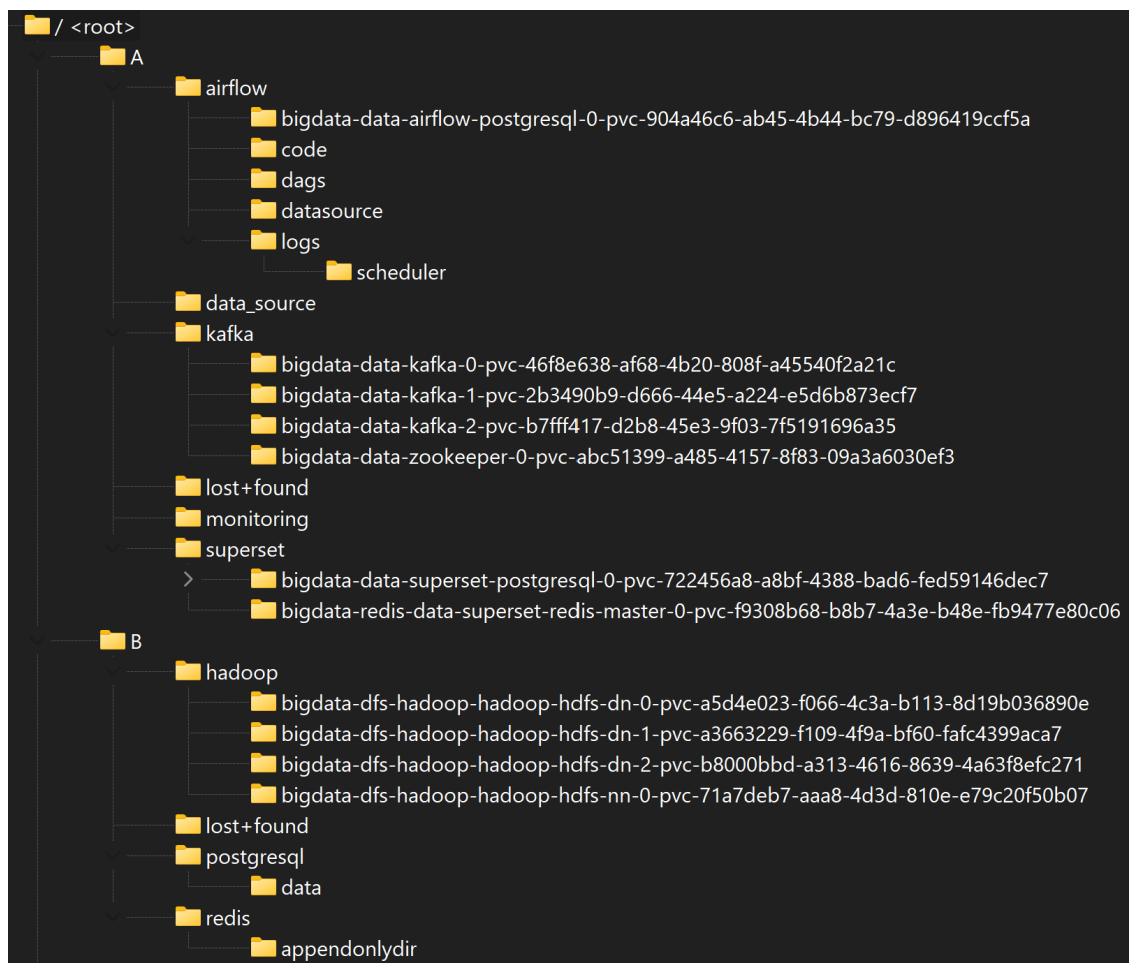
5.3.1 Triển khai cụm máy chủ

Từ giải pháp thiết kế kiến trúc hệ thống như trên, em đã triển khai được một cụm máy chủ phục vụ cho việc triển khai các ứng dụng lên môi trường Kubernetes On-Premise. Bảng 5.1 là thông số cấu hình cho các máy ảo trong cụm. Các máy ảo được kết nối với nhau qua mạng nội bộ, với tài nguyên được cung cấp sao cho phù hợp với cấu hình của máy host (ở đây là 64 GB RAM và 1TB ổ cứng). Sáu máy chủ chạy hệ điều hành Ubuntu bao gồm 3 k8s-master, Rancher Server, Load Blancer và NFS Server như đã trình bày ở phần 5.2

Bảng 5.1: Thông số các máy ảo trong cụm

Tên máy ảo	Địa chỉ IP	Số nhân CPU	RAM	Ổ cứng
k8s-master-1	192.168.164.201	8	12 GB	100 GB
k8s-master-2	192.168.164.202	8	12 GB	100 GB
k8s-master-3	192.168.164.203	8	12 GB	100 GB
rancher-server	192.168.164.204	2	2 GB	20 GB
load-balancer-k8s	192.168.164.205	2	2 GB	20 GB
nfs-server	192.168.164.206	2	2 GB	/A: 200 GB
				/B: 200 GB

5.3.2 Lưu dữ liệu các thành phần



Hình 5.5: Cấu trúc thư mục trên NFS Server

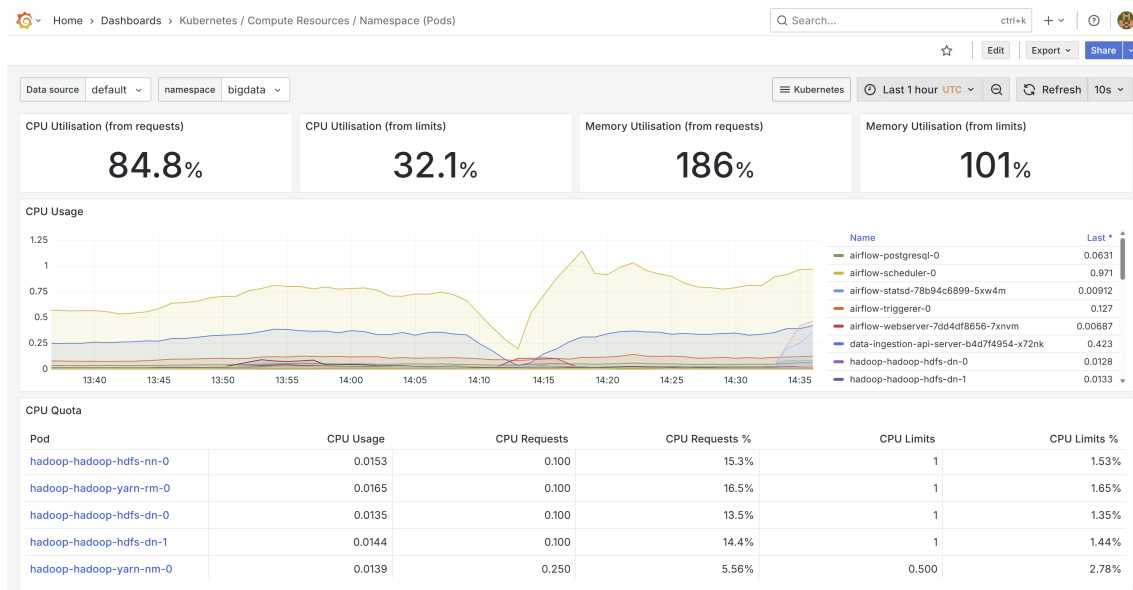
Hình 5.5 là kết quả của việc triển khai NFS Server để lưu trữ các PVC chứa dữ liệu của các ứng dụng. Ở đây em tạo 2 ổ đĩa riêng biệt có tên "/A" và "/B". Ổ đĩa A là nơi lưu trữ dữ liệu của Kafka, Airflow, dữ liệu nguồn phục vụ giả lập thu thập, Superset, Prometheus, dữ liệu sao lưu của hệ thống. Ổ đĩa B lưu trữ dữ liệu của các cơ sở dữ liệu như Hadoop HDFS, PostgreSQL, Redis. Mỗi ứng dụng có một thư mục riêng biệt của riêng mình, đảm bảo dữ liệu được đọc ghi một cách dễ dàng. Ngoài ra việc tách biệt lưu trữ ra một nơi khác cũng sẽ tối ưu hiệu suất hoạt động của các máy chủ chạy Kubernetes.

5.3.3 Công cụ giám sát & Sao lưu hệ thống

Ngoài các thành phần đã kể trên, em còn triển khai thêm các công cụ giám sát và sao lưu cụm Kubernetes để đảm bảo khả năng vận hành ổn định và khôi phục dữ liệu khi cần thiết.

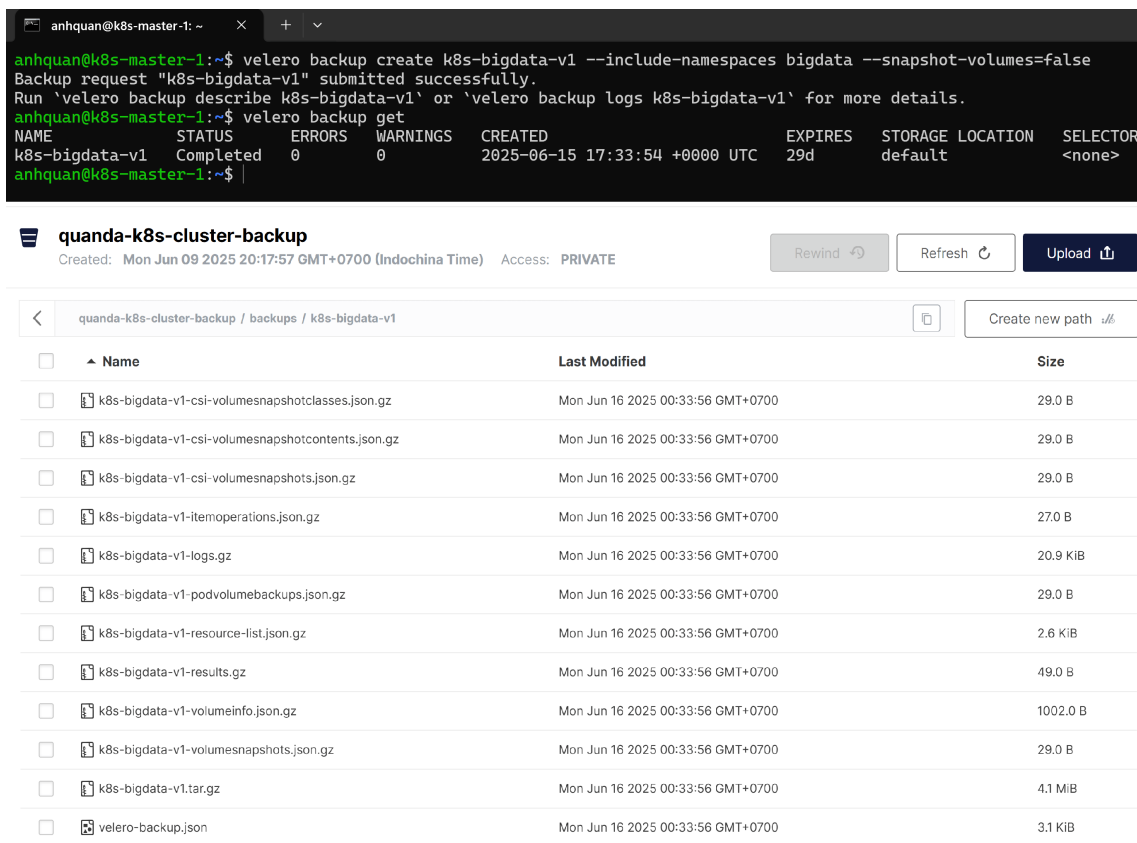
Prometheus là công cụ thu thập và lưu trữ metrics thời gian thực từ các node, Pod và dịch vụ trong hệ thống. Em triển khai Prometheus theo mô hình operator, cho

phép tự động khám phá các service endpoint trong cluster. Prometheus được cấu hình để scrape metrics từ node-exporter, kube-state-metrics và các exporter khác như spark/grafana/airflow khi có. Grafana được tích hợp cùng Prometheus để hiển thị dashboard trực quan theo thời gian thực. Grafana có khả năng tạo các dashboard tổng quan trạng thái cụm K8s (CPU, memory, pod, container), dashboard theo dõi từng dịch vụ: Kafka throughput, Spark jobs, PostgreSQL query, Redis cache... hay cảnh báo khi CPU hoặc bộ nhớ vượt ngưỡng cho phép, giúp sớm phát hiện sự cố. Hình 5.6 là ví dụ về dashboard giám sát tài nguyên của các Pod trong hệ thống.



Hình 5.6: Giao diện Grafana giám sát tài nguyên của các Pod

Để bảo vệ dữ liệu quan trọng và các cấu hình của hệ thống, em triển khai Velero – công cụ mã nguồn mở chuyên dụng để sao lưu và khôi phục cụm K8s. Velero được cấu hình sử dụng backend là hệ thống Object Storage MinIO, được triển khai cục bộ trên node lưu trữ. Velero cung cấp khả năng tạo ra các dữ liệu sao lưu theo namespace, có thể lưu trữ toàn bộ cụm hoặc các namespace cụ thể. Ngoài ra Velero còn cho phép tự động sao lưu theo lịch (hàng ngày, hàng tuần...) và khôi phục nhanh chóng toàn bộ cụm K8s về trạng thái trước đó. Hình 5.7 dưới đây là câu lệnh khởi tạo và cấu trúc thư mục chứa dữ liệu sao lưu của cụm K8s đối với namespace “bigdata” - là namespace triển khai toàn bộ các ứng dụng.



The image shows a terminal window and a web interface for a Kubernetes backup. The terminal window displays the command `velero backup create k8s-bigdata-v1 --include-namespaces bigdata --snapshot-volumes=false` and its output, indicating a successful backup. The web interface, titled "quanda-k8s-cluster-backup", shows the backup details for "k8s-bigdata-v1". It lists various backup files and their sizes.

NAME	STATUS	ERRORS	WARNINGS	CREATED	EXPIRES	STORAGE	LOCATION	SELECTOR
k8s-bigdata-v1	Completed	0	0	2025-06-15 17:33:54 +0000 UTC	29d	default		<none>

Name	Last Modified	Size
k8s-bigdata-v1-csi-volumesnapshotclasses.json.gz	Mon Jun 16 2025 00:33:56 GMT+0700	29.0 B
k8s-bigdata-v1-csi-volumesnapshotcontents.json.gz	Mon Jun 16 2025 00:33:56 GMT+0700	29.0 B
k8s-bigdata-v1-csi-volumesnapshots.json.gz	Mon Jun 16 2025 00:33:56 GMT+0700	29.0 B
k8s-bigdata-v1-itemoperations.json.gz	Mon Jun 16 2025 00:33:56 GMT+0700	27.0 B
k8s-bigdata-v1-logs.gz	Mon Jun 16 2025 00:33:56 GMT+0700	20.9 KiB
k8s-bigdata-v1-podvolumebackups.json.gz	Mon Jun 16 2025 00:33:56 GMT+0700	29.0 B
k8s-bigdata-v1-resource-list.json.gz	Mon Jun 16 2025 00:33:56 GMT+0700	2.6 KiB
k8s-bigdata-v1-results.gz	Mon Jun 16 2025 00:33:56 GMT+0700	49.0 B
k8s-bigdata-v1-volumeinfo.json.gz	Mon Jun 16 2025 00:33:56 GMT+0700	1002.0 B
k8s-bigdata-v1-volumesnapshots.json.gz	Mon Jun 16 2025 00:33:56 GMT+0700	29.0 B
k8s-bigdata-v1.tar.gz	Mon Jun 16 2025 00:33:56 GMT+0700	4.1 MiB
velero-backup.json	Mon Jun 16 2025 00:33:56 GMT+0700	3.1 KiB

Hình 5.7: Dữ liệu sao lưu cụm K8s lưu trữ trên MinIO

Việc tích hợp giám sát và sao lưu giúp hệ thống không chỉ vận hành hiệu quả, mà còn có thể phản ứng kịp thời khi xảy ra sự cố hoặc khôi phục trong các tình huống bất ngờ như lỗi hệ thống, mất điện hoặc sai sót cấu hình.

CHƯƠNG 6. KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

Kết thúc quá trình thiết kế, triển khai, kết quả thực nghiệm ở chương 4 và giải pháp đóng góp về triển khai môi trường Kubernetes On-Premise ở chương 5, chương này sẽ đưa ra những kết luận về hệ thống, đánh giá ưu, nhược điểm và đề xuất hướng phát triển để hoàn thiện sản phẩm.

6.1 Kết luận

Trong đồ án này, em đã tiến hành nghiên cứu và xây dựng một hệ thống xử lý dữ liệu lớn (Big Data) theo kiến trúc Lambda, phục vụ cho cả hai mục tiêu: phân tích tổng hợp (batch) và xử lý dữ liệu gần thời gian thực (streaming). Đồ án tập trung vào việc tích hợp toàn bộ các thành phần chính của một hệ thống dữ liệu lớn gồm thu thập, xử lý, lưu trữ, trực quan hóa và điều phối hệ thống. Đặc biệt, em lựa chọn triển khai toàn bộ hệ thống trên môi trường Kubernetes On-Premise nhằm kiểm soát tối đa hạ tầng, đảm bảo tính chủ động trong mở rộng, giám sát và phục hồi hệ thống.

Song song với quá trình triển khai, em cũng đã tiến hành thực nghiệm hệ thống trên tập dữ liệu taxi của thành phố New York để đánh giá hiệu quả xử lý, khả năng mở rộng và độ ổn định trong môi trường phân tán. Ngoài ra, em đã đề xuất giải pháp triển khai một cụm máy chủ Kubernetes trên môi trường hạ tầng nội bộ, sao cho phù hợp với những hệ thống nói chung và hệ thống dữ liệu lớn nói riêng.

Kết thúc quá trình thực hiện đồ án, em đã xây dựng thành công một hệ thống xử lý dữ liệu lớn với đầy đủ các chức năng: thu thập, xử lý, lưu trữ, trực quan hóa dữ liệu và quản lý giám sát hệ thống, đồng thời triển khai toàn bộ hệ thống trên môi trường Kubernetes On-Premise do em tự thiết lập từ đầu. Hệ thống đã cơ bản đáp ứng được các yêu cầu chức năng và phi chức năng được đề ra tại chương 2.

Về chức năng, hệ thống cho phép thu thập dữ liệu gần thời gian thực thông qua API, xử lý song song dữ liệu theo luồng và theo lô bằng kiến trúc Lambda, lưu trữ dữ liệu linh hoạt trên HDFS, PostgreSQL và Redis, và trực quan hóa dữ liệu qua các công cụ như Superset và giao diện Web. Hệ thống đảm bảo khả năng mở rộng, tự động hóa và tương tác giữa các mô-đun thông qua Kafka và dịch vụ nội bộ của Kubernetes.

So với các nghiên cứu và sản phẩm tương tự như hệ thống xử lý dữ liệu Spark-Kafka truyền thống chạy trên máy ảo hoặc dịch vụ cloud, đồ án này nổi bật ở việc triển khai mô phỏng trên hạ tầng vật lý thật (On-Premise Kubernetes), giúp kiểm soát linh hoạt các lớp tài nguyên (networking, storage, scheduler), nâng cao hiệu

biết thực tiễn về triển khai hệ thống phân tán.

Tuy nhiên, đồ án vẫn còn một số hạn chế. Cụ thể, hệ thống chưa xử lý được đầy đủ các tình huống lỗi phức tạp như mất kết nối mạng, tắt đột ngột node trong cụm K8s; mô hình dự đoán đơn giản, chưa áp dụng các phương pháp học sâu hoặc học tăng cường; giao diện Web còn ở mức cơ bản. Ngoài ra, việc tối ưu hóa hiệu suất Spark và bảo mật kết nối chưa được tập trung khai thác sâu.

6.2 Hướng phát triển

Mặc dù hệ thống đã hoàn thiện cơ bản các chức năng thu thập, xử lý, lưu trữ và trực quan hóa dữ liệu lớn, vẫn còn nhiều tiềm năng để tiếp tục phát triển nhằm nâng cao hiệu quả và khả năng ứng dụng thực tiễn.

Trước hết, cần hoàn thiện các chức năng hiện tại bằng cách tối ưu hiệu suất xử lý của Spark và Kafka, tăng cường khả năng chịu lỗi thông qua cấu hình pod phù hợp và backup dữ liệu, cũng như bảo mật hệ thống bằng các giải pháp quản lý secret, phân quyền và giám sát truy cập. Việc tự động hóa quá trình triển khai bằng Helm Chart thay vì YAML thủ công cũng là một bước quan trọng để tăng tính linh hoạt và tái sử dụng.

Bên cạnh đó, hệ thống có thể được mở rộng theo nhiều hướng như tích hợp các mô hình học máy nâng cao để cải thiện khả năng dự đoán, triển khai thêm hệ thống giám sát – cảnh báo sử dụng Prometheus và Grafana, xây dựng pipeline CI/CD để hỗ trợ triển khai đa môi trường (Dev, Test, Prod), cũng như bổ sung thêm các nguồn dữ liệu như thời tiết, sự kiện giao thông để phục vụ phân tích đa chiều.

Cuối cùng, việc triển khai hệ thống lên các nền tảng đám mây thực tế như AWS, GCP hoặc Azure sẽ giúp khai thác tối đa lợi thế mở rộng linh hoạt, từ đó đưa hệ thống đến gần hơn với các ứng dụng sản xuất quy mô lớn.

TÀI LIỆU THAM KHẢO

- [1] J. Lin, “The lambda and the kappa,” *IEEE Internet Computing*, vol. 21, no. 05, pp. 60–66, 2017.
- [2] G. K. Kalipe and R. K. Behera, “Big data architectures: A detailed and application oriented review,” *Int. Journal Innov. Technol. Explor. Eng.*, vol. 8, pp. 2182–2190, 2019.
- [3] M. Gaianu, “On premise data center vs cloud,” in *2023 International Conference on Computational Science and Computational Intelligence (CSCI)*, IEEE, 2023, pp. 1068–1071.
- [4] Atlan, *Cloud vs on-premise vs hybrid: Which one is best for you?* [Online]. Available: <https://atlan.com/cloud-vs-on-premise-vs-hybrid/> (visited on 05/07/2025).
- [5] Apache Software Foundation, *Apache airflow*. [Online]. Available: <https://airflow.apache.org/docs/apache-airflow/stable/index.html> (visited on 05/08/2025).
- [6] Apache Software Foundation, *Apache kafka*. [Online]. Available: <https://kafka.apache.org/documentation/> (visited on 05/08/2025).
- [7] Apache Software Foundation, *Apache spark*. [Online]. Available: <https://spark.apache.org/docs/latest/> (visited on 05/08/2025).
- [8] Apache Software Foundation, *Hdfs architecture*. [Online]. Available: <https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html> (visited on 05/08/2025).
- [9] Redis, *Redis*. [Online]. Available: <https://redis.io/docs/latest/> (visited on 05/08/2025).
- [10] The PostgreSQL Global Development Group, *Postgresql*. [Online]. Available: <https://www.postgresql.org/docs/> (visited on 05/08/2025).
- [11] Apache Software Foundation, *Superset*. [Online]. Available: <https://superset.apache.org/docs/intro/> (visited on 05/08/2025).
- [12] VMWare Tanzu, *Spring boot*. [Online]. Available: <https://spring.io/projects/spring-boot> (visited on 05/08/2025).
- [13] MetaOpenSource, *React*. [Online]. Available: <https://react.dev/> (visited on 05/08/2025).
- [14] The Prometheus Authors, *Prometheus overview*. [Online]. Available: <https://prometheus.io/docs/introduction/overview/> (visited on 05/08/2025).

- [15] Grafana Labs, *Technical documentation*. [Online]. Available: <https://grafana.com/docs/> (visited on 05/08/2025).
- [16] The Kubernetes Authors, *Kubernetes documentation*. [Online]. Available: <https://kubernetes.io/docs/home/> (visited on 05/08/2025).
- [17] L. Breiman, “Random forests,” *Machine learning*, vol. 45, pp. 5–32, 2001.
- [18] Velero Authors, *Backup and migrate kubernetes resources and persistent volumes*. [Online]. Available: <https://velero.io/> (visited on 06/15/2025).
- [19] MinIO, Inc, *Minio*. [Online]. Available: <https://min.io/> (visited on 06/15/2025).
- [20] DEVOPSEDU.VN, *Khóa học kubernetes thực tế*. [Online]. Available: <https://devopsedu.vn/courses/khoa-hoc-kubernetes-thuc-te/> (visited on 03/07/2025).
- [21] Rancher, *Rancher*. [Online]. Available: <https://www.rancher.com/> (visited on 05/24/2025).